Towards Synchronizing Models with Evolving Metamodels

Boris Gruschko
SAP Research
CEC Karlsruhe
Vincenz-Priessnitz-Strasse 1
76131 Karlsruhe, Germany
boris.gruschko@sap.com

Dimitrios S. Kolovos
Department of Computer Science
The University of York
York, UK, YO10 5DD
dkolovos@cs.york.ac.uk

Richard F. Paige
Department of Computer Science
The University of York
York, UK, YO10 5DD
paige@cs.york.ac.uk

Abstract

Metamodel evolution poses a threat to the applicability of Model-Driven Development to large scale projects. The problem is caused by incompatibilities between metamodel revisions. These render models that conform to the older version of the metamodel non-conformant to the newer version. An approach to addressing this problem is co-evolution of models with their respective metamodels. In this paper we introduce the problem of synchronizing models with evolving metamodels and outline an approach to addressing it efficiently. The aim of the proposed approach is to minimize the effort required to perform model migration in face of metamodel changes. To provide deeper insights into the envisioned approach, we demonstrate preliminary solutions to the problem of change detection between two metamodel revisions. Furthermore, we present an approach to model-to-model transformations, using a conservative copying algorithm, which regulates the retainment of instances during model migration.

1 Introduction

Metamodeling is the cornerstone of Model-Driven Development (MDD). Metamodels are used to formalize the artefacts of the MDD process. Like all software artefacts, metamodels typically *evolve* during a software development project, due to the progressively enhanced understanding of the problem domain and how it should be modelled. However, changes introduced when evolving a metamodel can invalidate the models that conform to its previous version.

1.1 Metamodeling in general

The Meta Object Facility (MOF) is a metamodelling architecture that supports rigorous definition of modelling languages. MOF introduces four abstraction levels (numbered M3-M0 [10]). M3 is the meta-meta-model, used to describe metamodels; it is self-describing. M2 contains metamodels, which are instances of the M3 model. M1 holds instances of M2 models. Finally, M0 represents the instances of M1 models. The M0 level corresponds to actual datasets and is out of the scope of this paper. The rest of this paper follows this OMG terminology.

M2 models are authored using the constructs provided by the M3 model. Two of the most widely used M3 models are ECore [4] and MOF [10, 11]. In this paper we consider M2 models constructed as instances of ECore only. However, the presented ideas also apply for other modelling technologies.

1.2 Problem Description

M1 model erosion is caused by changes in the corresponding M2 model. M2 models can change in several ways. Some of these changes are additive, and as such they do not break the corresponding M1 models. Other changes introduce incompatibilities and cross-version inconsistencies, therefore invalidating (or "breaking") M1 models. As a consequence, the broken M1 models cannot be imported by the modelling infrastructure, which is aware of the newer

M2 model version only. To address this issue, the M1 instances of the changed M2 model have to be *migrated* to the newer M2 model version. The efforts needed to perform this migration are dependent on the extent and nature of changes made to the M2 model.

1.3 Introductory Example

In this section, we provide a concrete example of metamodel evolution to illustrate the problem of M1 model erosion. The M2 model we use for this purpose represents the structure of a file system. The first iteration of M2 model creation delivered the M2 model depicted in Figure 1.

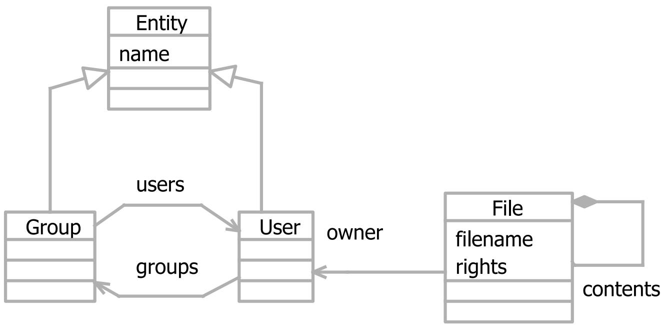

Figure 1. First version of M2 model

After further analysis, the proposed file system M2 model is changed. The newer version of the M2 model is depicted in Figure 2. The most significant change in the newer version is splitting the $File$ class into four new classes. The access permission handling has been moved into its own $Rights$ class. Furthermore, the container $Directory$ class has been created. Finally, the $FileSystemElement$ class has been introduced, to enable uniform ownership and access control permission handling for both files and directories. Although, the provided description captures the most significant changes, there are also a number of minor changes which compromise M1 model validity.

The particular changes introduced into the file system model are as follows:

- Class $Entity$ renamed to $NamedElement$.
- Reference between $Group$ and $User$ classes renamed from $users$ to $members$.
- New classes $Directory$, $FileSystemElement$, and $Rights$ have been added.
- Attributes have been added to classes $Rights$ and $File$. In particular, attributes $canRead$, $canWrite$, $canExecute$ have been added to $Rights$ and $filename$, $rights$, and $extension$ have been added to $File$.
- New subclassing has been added: $Directory$ and $File$ (subclasses of $FileSystemElement$) and $FileSystemElement$ subclasses $NamedElement$.
- Containment references have been moved or created: contents has been moved from $File$ to connect $Directory$ and $FileSystemElement$. Similarly, $otherRights$, $userRights$, and $groupRights$ containments have been added in Figure 2.

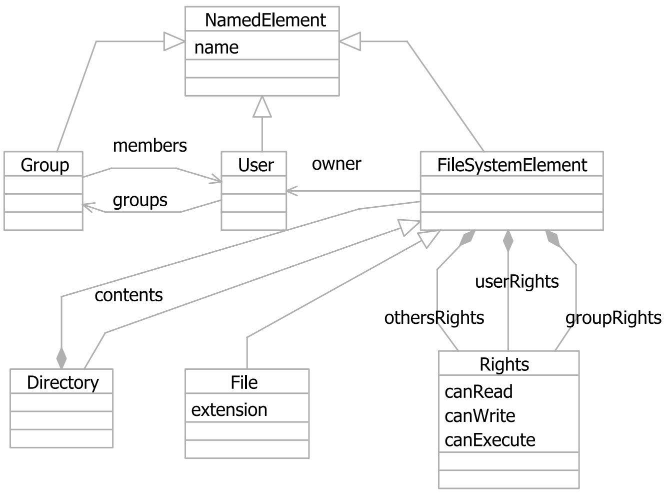

Figure 2. Second version of M2 model

This list does not capture the intention behind the changes. For example, the values of the $Rights$ class instances could be derived from the old values of the $rights$ attributes from the corresponding $File$ instances.

1.4 Outline

The rest of this paper is structured as follows. Section 2 provides an overview and initial classification of possible ways in which M2 models may change. The intention of this section is not to provide an exhaustive list, but to illustrate a number of examples encountered in practice. Section 3 describes the envisioned approach to addressing the migration problem. In Section 4 we describe our experiences with an early prototype of the envisioned migration system and provide an overview of the future directions. Section 5 provides links to works related to the presented problem. In Section 6 we conclude and provide directions to further research on the subject.

2 A Classification of M2 model changes

This section provides an overview of frequently encountered changes in M2 models. We propose a classification into following three categories:

- **Not Breaking Changes**: changes which occur in the M2 model, but don't break its instances.
- **Breaking and Resolvable Changes**: changes which do break the M2 model instances, but can be resolved by automatic means.
- **Breaking and Unresolvable Changes**: changes which do break the M2 model instances and can not be resolved automatically.

Changes in the first category are not relevant to the migration problem. Changes placed into the second category can be resolved without user intervention. It is our goal to make the migration of M2 model instances affected by these changes seamless to the operators of the migration system. The third category of changes requires human attention. The operator will have to provide additional information needed to migrate the affected instances.

2.1 Renames

Renames refer to changes affecting the names of M2 model elements. In the particular case of Ecore based M2 models, the changes are affecting the $name$ attribute of the $ENamedElement$ class. Because most of Ecore classes (with the exception of $EAnnotation$ and $EFactory$) are specializations of $ENamedElement$, changes affecting this attribute have an impact on almost all M1 instances as far, as the M2 classes are referred to by their names in M1 instances.

The rename changes should be automatically resolvable, if considered in isolation. Ecore relies on the hierarchical package structure for namespace maintenance. Therefore, it is possible to build a bijective function, reflecting the renames on package contents. Due to the disjunctive nature of package contents, the evaluation of the constructed function will not be affected by functions corresponding to other packages. It is to be evaluated, how the renaming functions behave, if correlated with other structural changes.

2.2 Deletion of M2 Objects

Deleting an object from the M2 level may mean that the information it captures is now obsolete and consequently its instances should also be removed from M1 models. On the other hand the act of deleting an object may be part of a larger-scale activity (e.g. a refinement). In this case, information from the instances of the now deleted M2 object may need to be stored elsewhere in the M1 models.

The deletion of M2 objects requires intervention, in order to delete the corresponding M1 instances. The M1 model will be invalidated, if the M1 instances of the deleted M2 objects are retained in the M1 model.

2.3 Addition of M2 Objects

The addition of M2 objects is an additive change. Therefore, no intervention during the migration is needed. Exceptions can occur if the M2 model contains constraints which explicitly disallow the presence of certain M2 objects.

2.4 Movement of M2 Objects

At the M2 level, movement of an M2 object amounts to a deletion of one and creation of another M2 object with the same type and attribute values. However, if the M1 instances of the concerned M2 object are considered, this solution implies the loss of all existing M1 instances of the moved object. To avoid this, the M1 instances of the concerned M2 object have to be migrated, to refer to the new M2 object location.

For movements of M2 objects, a distinction between movements which do not change the number of M1 instances and those changing the number of M1 instances has to be made. The former requires the translation of links to the M2 objects only. An example of such change is the movement of an M2 class to another package. During this transition the number of M1 instances remains intact. Changes demanding the alteration of the number of M1 instances require a more complex migration process. An example of an M2 change, requiring the alternation of the number of M1 instances, is the movement of an attribute from one M2 class to another M2 class. This change, will require the migration process to decide, which attribute values should be retained in the M1 model after migration and which instances these values should be assigned to.

2.5 Associations changes

Changes concerning the associations of two M2 classes tend to be either additive, or breaking and not-resolvable. An example for the former case could be the extension of the cardinality interval on a reference. if the cardinality is being extended (e.g. from $1..n$ to $0..n$), then the corresponding M1 models will not be broken. Thus, no intervention is required, independently of the contents of M1 models. However, if the cardinality is being restricted (e.g. from $0..n$ to $1..n$), then every link set in the M1 model with zero links will become invalid. This change can not be resolved automatically, because a human operator has to decide about the creation of new links.

A similar scenario can be observed when changing a reference from containment to simple reference, and vice versa. While the change from containment to simple reference is additive, the change from simple reference to containment may be breaking and unresolvable. Breaking during simple reference to containment transition results from the possible multiple containment of a single M1 object after the transition to the new M2 model version.

2.6 Type changes

Type changes, are changes of attribute types. To allow for a simpler treatment of these changes, there is a distinction to be made between primitive and complex typed attributes. For primitive types, a mapping can be created, to allow for an automatic migration. For instance, while a change from $Integer$ to $String$ attribute can not produce invalid results, a reversed change can produce some invalid conversions, which have to be addressed manually. The problem of complex typed attribute migration is not considered at this point in time.

3 Proposed approach

The described problem of M1 model migration can, in our view, be split into various sub-tasks. Firstly, the differences between two M2 model versions need to be determined. Secondly, an appropriate algorithm for model migration has to be determined. Finally, the determined migration algorithm is to be executed, obtaining valid M1 models for the newer M2 model version.

We propose an approach to the M1 model migration, with emphasis on the minimization of manual effort. Figure 3 provides an overview of the propsed approach. The envisioned steps are:

1. Change detection or tracing: The M2 model versions are compared and the differences are translated into the delta model.

2. Classification: The found changes are classified into categories described in Section 2.

3. User input gathering: The user input needed for not automaticaly resolvable changed migration is gathered.

4. Algorithms determination: The algorithms required for the migration are determined.

5. Migration: The migration is executed.

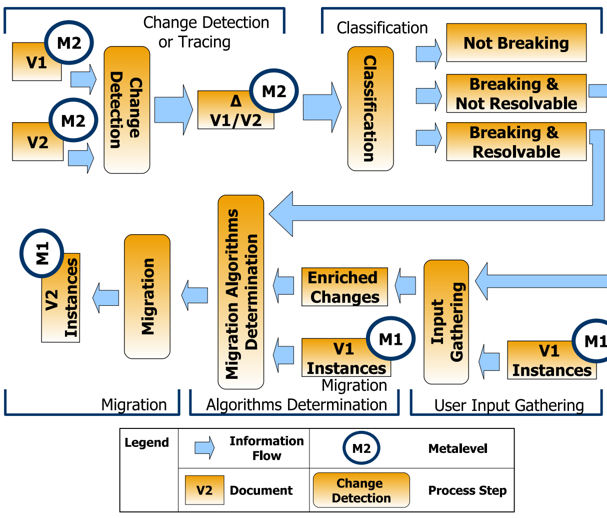

Figure 3. Envisioned migration process.

In this paper we concentrate on two of the above mentioned topics, namely the detection of changes between two M2 model revisions and the migration of unresolvable changes. The remaining topics are the subject of future research.

3.1 Change detection

The product of change detection is a set of changes performed on a M2 model. In our view, there are two methods to obtain this set of changes. The first obtains two versions of the M2 model as input and delivers the set of changes between them. In the sequel we refer to this method as *direct comparison*. The second method takes the older version of the M2 model and a trace of change operations as input, in order to deliver the same output as the first method. Both methods have advantages and disadvantages. The direct comparison can work with any modelling tool, without requiring a change trace recording capability. However, the algorithms required to perform the comparison are more complex and will not always deliver the desired result. The changes trace method requires a modelling tool capable of keeping track of changes performed on the M2 model. However, this method provides the capability to track, for example, model element moves in absence of unique element identifiers. The set of changes obtained via one of the presented methods is to be maintained as an instance of the change M2 model.

3.1.1 Change Metamodel

The Eclipse Modeling Framework provides a change metamodel, designed to capture changes on instances of Ecore-based M2 models. Although useful for our purposes, the

EMF Change metamodel is in our view insufficient for the purpose of M2 model changes tracking. An instance of EMF's change metamodel provides a set of element and attribute changes, without capturing their semantic. For instance, a change of the attribute $name$ on a structural feature would me the renaming of an attribute in the containing class. The correspondence between the actual attribute change and the rename operation is not being captured by EMF's Change metamodel. In the envisioned metamodel, we would like to have a class $Rename$ with instances indicating the particular renames of model elements. This, more constrained structure would allow for simpler construction of the following migration steps, like change classification and M1 model migration.

3.1.2 Direct Comparison

Direct comparison of two M2 models is a task similar to the comparison of UML models. In our opinion, the comparison of two M2 models would only yield a useful result when model elements possess unique identifiers. This is the case for MOF-based M2 models, since the $mofid$ attribute is mandatory for all M2 model elements. However, the Ecore-based M2 models are not required to have uniquely identifiable elements. This makes the detection of element moves impossible, because an element move would be indistinguishable from an element deletion followed by an element creation.

3.1.3 Changes trace

The creation of the changes trace requires modifications on modeling tools used to author the M2 model. To verify the feasibility of this approach, we modified the EMF modelling tool set, to export traces of changes performed on Ecore-based M2 models. Due to the availability of a built-in $ChangeRecorder$ facility in the EMF tool set, this can be achieved with a minimal development effort.

3.2 Model Migration/Transformation

In the absense of a change trace, or for non-resolvable changes, a transformation can be composed to transforms instances of the old M2 model into instances of the updated M2 model. The current trend in Model Transformation is to use rule-based languages such as QVT [9], ATL [6] and ETL [5, 1]. The main advantage of using one of those transformation languages is that the transformation logic can be expressed at a high level of abstraction thus enhancing maintainability and understandability. To express a transformation in such a language, the user must specify mapping rules that describe how model elements from the input model can be mapped to elements in the target model. For example, an ETL transformation that could perform this task is the following [1]:

Listing 1. Complete transformation for migrating models of the evolved FS metamodel specified in ETL

```
import 'FS2Utils.eol';

rule FileSystem2FileSystem
   transform s : FS1!FileSystem
   to t : FS2!FileSystem {

   t.contents ::=
      s.entities.includingAll(s.files);
}

rule User2User
   transform s : FS1!User
   to t : FS2!User {

   t.name := s.name;
   t.groups ::= s.groups;
}

rule Group2Group
   transform s : FS1!Group
   to t : FS2!Group {

   t.name := s.name;
   t.members ::= s.users;
}

abstract rule File2FileSystemElement
   transform s : FS1!File
   to t : FS2!FileSystemElement {

   t.userRights := s.rights
      .getDigit(0).toRightsElement();
   t.groupRights := s.rights
      .getDigit(1).toRightsElement();
   t.othersRights := s.rights
      .getDigit(2).toRightsElement();
   t.parent ::= s.parent;
   t.owner ::= s.owner;
}

rule File2File
   transform s : FS1!File
   to t : FS2!File
   extends File2FileSystemElement {

   guard : s.contents.size() = 0

   t.name := s.filename.getFilename();
   t.extension :=
      s.filename.getExtension();
}

rule File2Directory
   transform s : FS1!File
   to t : FS2!Directory
```

[1]The contents of the FS2Utils.eol file imported by the transformation are presented in the Appendix

```
    extends File2FileSystemElement {

    guard : s.contents.size() > 0

    t.name := s.filename;
    t.contents ::= s.contents;
}
```

By inspecting this transformation it becomes apparent that it contains a significant amount of trivial information. For instance, since the $User$ entity has not changed at all, the $User2User$ rule simply performs a copy of instances of User in the source model to instances of User in the target model. Since our metamodel is quite small and the changes from the first to the second are significant, in this case there is only one case of such a trivial transformation rule. However, for M2 models containing hundreds of metaclasses (e.g. the UML2.0 metamodel), manually writing an equivalent number of trivial rules is inefficient both in terms of understanding and maintainability. Interestingly, ATL supports a refinement mode which can be used in case the source and target models are of the same metamodel, to eliminate the need for writing such trivial rules. In this mode, if no rule is specified for an element in the source model, the element is simply copied into the target model. However, in the case of model migration the source and target models are not of the same metamodel but of *very similar* metamodels. Therefore, instead of an exact copy mechanism (such as that implemented by the ATL refinement mode), it would be desirable to implement a *conservative copy* mode which can however be extended and customized using user-defined transformation rules. Unfortunately, implementing such a mode for ATL is not possible without changing the syntax of the language. By contrast, the Epsilon Transformation Language (ETL) provides support for such pluggable transformation algorithms (strategies). Therefore, we have implemented the aforementioned conservative copy algorithm as a strategy for ETL.

3.3 Use of Strategies in ETL

In an ETL transformation module, a transformation strategy can be attached to reduce the number of hand-written rules. A strategy effectively encodes recurring patterns of transformation. A transformation strategy is invoked in two cases. In case of absence of a rule that can transform an element from the source model, the element is passed to the strategy by calling its $autoTransform$ method, passing a source object and a context to the method. This method is then responsible for creating and populating any instances in the target model. The other case when the transformation strategy is invoked is when an $auto$ rule is encountered. When an $auto$ rule is executed, the ETL engine creates the target elements that the rule defines and then passes them on to the strategy by calling its $autoTransform$ method, passing additionally a collection of targets. This method is then responsible only for populating the already created targets. Once the $autoTransform$ method has been executed, the engine also executes the body of the rule.

3.4 The ConservativeCopyStrategy

3.4.1 The autoTransform(Object source, EtlContext context) method

This method is called when no rule has been found for the source element. When it is invoked it attempts to create an element of a metaclass with the same name in the target model. If the target metamodel does not contain a metaclass with this name, the method fails silently without producing runtime errors. Otherwise, it creates the new elements and calls the *autoTransform(Object source, Collection targets, EtlContext context)* method with targets being a collection containing only the newly created element.

3.4.2 The autoTransform(Object source, Collection targets, EtlContext context) method

This method is called either when an $auto$ rule is encountered or from the *autoTransform(Object source, EtlContext context)* method as discussed above. This method performs a conservative copy of the features of the source element into the first element of the targets collection. More specifically, for each feature of the source element, it tries to find a feature with the same name in the target element, if the feature is found there are two cases. If the source and the target features are of the same primitive types, the value of the feature is copied. If both the source and the target feature are of user-defined types the equivalent value of the source-feature value (which is calculated recursively by invoking other rules and/or the strategy itself) is assigned as the value of the target feature. In case of type mismatch, or any other error, the method fails silently and proceeds with the next feature of the source element.

3.5 Implementing the Transformation Using the ConservativeCopyStrategy

Having specified the functionality of the $CosnervativeCopyStrategy$, in this section we rewrite the same transformation using ETL.

Listing 2. The transformation of Listing 1 simplified using the $ConservativeCopyStrategy$ strategy

```
import 'FS2Utils.eol';
```

```
rule FileSystem2FileSystem
   transform s : FS1!FileSystem
   to t : FS2!FileSystem {

   t.contents ::= s.entities
      .includingAll(s.files);
}

auto rule Group2Group
   transform s : FS1!Group
   to t : FS2!Group {

   t.members ::= s.users;
}

abstract auto rule File2FileSystemElement
   transform s : FS1!File
   to t : FS2!FileSystemElement {

   t.userRights := s.rights
      .getDigit(0).toRightsElement();
   t.groupRights := s.rights
      .getDigit(1).toRightsElement();
   t.othersRights := s.rights
      .getDigit(2).toRightsElement();
}

auto rule File2File
   transform s : FS1!File
   to t : FS2!File
   extends File2FileSystemElement {

   guard : s.contents.size() = 0

   t.name := s.filename.getFilename();
   t.extension :=
      s.filename.getExtension();

}

auto rule File2Directory
   transform s : FS1!File
   to t : FS2!Directory
   extends File2FileSystemElement {

   guard : s.contents.size() > 0

   t.name := s.filename;
}
```

In this transformation, rules that simply copy model elements from the source to the target model do not have to be hand-written since this functionality is implemented by the $ConservativeCopyStrategy$. For instance, this transformation does not contain the $User2User$ rule of Listing 1. Moreover, the specification each rule needs to provide is limited to the changed aspects of the metamodel. Therefore in the $File2FileSystemElement$ rule, unlike the transformation of Listing 1, the trivial $t.parent$ $::=$ $s.parent$; and $t.owner$ $::=$ $s.owner$; statements can be omitted. Although for the purpose of brevity our metamodels are limited in size, for large metamodels omitting trivial rules and statements greatly reduces the size of the migration transformation.

4 Early Experiences and Future Directions

To study the feasibility of the proposed approach, we created a vertical prototype of the envisioned system. The prototype provides the possibility to change M2 models, record the changes and then migrate the existing M1 models from older to the newer revision of the M2 model. The only type of changes the created prototype is able to migrate are renames of structural features.

The work flow in the created prototype consists of following steps:

1. Load the M2 model to be change into the M2 model editor.
2. Start the change recorder.
3. Perform the desired changes on the M2 model.
4. Stop the recorder.
5. Export the changes trace.
6. Link the exported changes trace to the M1 model instantiator.
7. Execute instantiation.

Prior to the instantiation of the M1 model to be migrated, the changes trace has to be interpreted to allow for the migrator construction. In our use case, where only the renames of structural features are being handled, the migrator consists of a lookup table filled with correspondences between old and new structural features names. The migration of the M1 model takes place during M1 model instantiation.

This simple prototype helped us demonstrate that it is feasible to record changes made to an M2 model, determine changes relevant to migration, construct an algorithm for migration, and execute the migration. Still to be investigated is possibility of embedding manual migration steps outlined in section 3.2 in this process.

5 Related Work

This section provides a brief overview of the existing work relevant to the proposed research topics. This review is by no means complete, due to the early stages of our research.

The modelling field has a large number of standards associated with it. Due to our focus on a fixed M3 model, existing M3 models are of particular interest to us. In this field, there exists an acknowledged OMG standard MOF [10, 11], which recently underwent a revision. However, in this paper we concentrated on the Eclipse Modeling Framework [4] and its M3 model, the Ecore, as a de facto industry standard. EMF does not have a written specification and is therefore defined in terms of deployable software artefacts.

The general field of model management has been defined in [3] under the term of *Generic Model Management*. This field has been developed by Melnik in [8]. It is our intention to embed our research in the field of Generic Model Management. Further work on the general field of model migration has been performed by Sprinkle in [14]. The classification of M2 model changes and the visualization aspects developed by Sprinkle in [15] are of significance to the changes classification and user input gathering stages of our approach.

There are no industrial standards concerning the topic of M1 model migration. The OMG has developed the MOF 2.0 versioning specification [12]. This standard is concerned with attaching versioning information to M2 models, but fails to provide means to obtain the versioning information. The M1 models are not in scope of this standard.

The comparison of M2 models closely resembles the problem of UML model comparison. One algorithm suitable for this comparison is UMLDiff$_{cld}$, proposed by Girschick in [7]. A more general approach to the difference creation between two models is given by Alanen and Porres in [2]. These approaches are suitable for the direct comparison option. The changes trace approach is closely related to the problem of information loss in versioning systems, described by Robbes and Lanza in [13].

6 Conclusion

The difficulty of model migration poses a serious threat to the applicability of MDD in the context of large scale software engineering. Due to the cross-metalevel correspondences between the MDD artifacts, state of the art revisioning systems are failing to provide a solution for this problem.

In this paper we have presented a generic approach to addressing the model migration problem. To gain early experiences with the envisioned approach, we implemented a transformation for M1 models, compensating for M2 model changes. This step provided a feasibility study for unresolvable changes migration. Further, we implemented an early vertical prototype of the migrator for resolvable changes. These experience indicate that the envisioned approach is capable of delivering on the promise of minimal effort M1 model migration.

References

[1] Epsilon component - Eclipse Generative Modeling Technology (GMT), Official Web-Site. http://www.eclipse.org/gmt/epsilon.

[2] Marcus Alanen and Ivan Porres. Differences and union of models. In Perdita Stevens, Jon Whittle, and Grady Booch, editors, *UML 2003 - The Unified Modeling Language. Model Languages and Applications. 6th International Conference, San Francisco, CA, USA, October 2003, Proceedings*, volume 2863 of *LNCS*, pages 2–17. Springer, 2003.

[3] Philip A. Bernstein, Laura M. Haas, Matthias Jarke, Erhard Rahm, and Gio Wiederhold. Panel: Is generic metadata management feasible? In Amr El Abbadi, Michael L. Brodie, Sharma Chakravarthy, Umeshwar Dayal, Nabil Kamel, Gunter Schlageter, and Kyu-Young Whang, editors, *VLDB*, pages 660–662. Morgan Kaufmann, 2000.

[4] Frank Budinsky, David Steinberg, Ed Merks, Raymond Ellersick, and Timothy Grose. *Eclipse Modeling Framework*. Addison Wesley Professional, 2003.

[5] Dimitrios S. Kolovos, Richard F. Paige, Fiona A.C. Polack. Eclipse Development Tools for Epsilon. In *Eclipse Summit Europe, Eclipse Modeling Symposium*, Esslingen, Germany, October 2006.

[6] Frédéric Jouault and Ivan Kurtev. Transforming Models with the ATL. In Jean-Michel Bruel, editor, *Proceedings of the Model Transformations in Practice Workshop at MoDELS 2005*, volume 3844 of *LNCS*, pages 128–138, Montego Bay, Jamaica, October 2005.

[7] Martin Girschick. Difference Detection and Visualization in UML Class Diagrams. Technical Report TUD-CS-2006-5, TU Darmstadt, 2006.

[8] Sergey Melnik. *Generic Model Management: Concepts and Algorithms*, volume 2967 of *Lecture Notes in Computer Science*. Springer, 2004.

[9] Object Management Group. MOF QVT Final Adopted Specification. http://www.omg.org/cgi-bin/doc?ptc/05-11-01.pdf.

[10] Object Management Group. *Meta Object Facility (MOF) Specification — Version 1.4*, April 2002.

[11] Object Management Group. *Meta Object Facility (MOF) 2.0 Core Final Adopted Specification*, 2004.

[12] Object Management Group. *MOF2 Versioning Final Adopted Specification*, 2005.

[13] Romain Robbes and Michele Lanza. Change-based software evolution. In *Proceedings of EVOL 2006 (1st International ERCIM Workshop on Challenges in Software Evolution)*, pages 159–164, 2006.

[14] Jonathan Sprinkle. *Metamodel Driven Model Migration*. PhD thesis, Vanderbilt University, Nashville, TN 37203, August 2003.

[15] Jonathan Sprinkle and Gabor Karsai. A domain-specific visual language for domain model evolution. *J. Vis. Lang. Comput*, 15(3-4):291–307, 2004.

7 Appendix: Operations defined in FS2Utils.eol

Listing 3. FS2Utils.eol

```
operation Integer getDigit(index : Integer) : Integer {
      return self.asString().toCharSequence()
            .at(index).asInteger();
}

operation Integer toRightsElement() : FS2!Rights {
      var rights : new FS2!Rights;
      if (self > 0) {
            rights.canRead := true;
      }
      if (self > 2) {
            rights.canWrite := true;
      }
      if (self > 6) {
            rights.canExecute := true;
      }
      return rights;
}

operation String getFilename() : String {
      var parts : Sequence;
      var filename : String;
      parts := self.split('\\.');
      if (parts.size() = 1) {return self;}
      for (p in parts) {
            if (hasMore) {
                  filename := filename + p;
            }
      }
      return filename;
}

operation String getExtension() : String {
      var parts : Sequence;
      parts := self.split('\\.');
      if (parts.size() > 0) {
            return parts.at(parts.size() - 1);
      }
      else {
            return '';
      }
}
```