Model Comparison: A Foundation for Model Composition and Model Transformation Testing

Dimitrios S. Kolovos
Department of Computer Science
The University of York
YO105DD, York, UK
dkolovos@cs.york.ac.uk

Richard F. Paige
Department of Computer Science
The University of York
YO105DD, York, UK
paige@cs.york.ac.uk

Fiona A.C. Polack
Department of Computer Science
The University of York
YO105DD, York, UK
fiona@cs.york.ac.uk

ABSTRACT

In the context of Model Driven Development, Model Transformation and Model Composition are two essential model management tasks. In this paper, we demonstrate how both tasks can benefit, in different ways, from the automation of another fundamental task: Model Comparison. We derive requirements for a model comparison solution incrementally, and demonstrate a concrete rule-based model comparison approach we have developed in the context of a generic model merging language.

Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]: Model checking, Validation

General Terms

Languages, Design

Keywords

Model Comparison, Model Transformation Testing, Model Composition, Model Driven Development

1. INTRODUCTION

A typical Model Driven Development process involves a number of models, often defined in different languages (metamodels) and technological platforms. As MDD raises models into first-class artefacts, they constitute live entities, manageable in diverse ways. For example, models can be transformed into other models, textual artefacts such as code and documentation can be generated from them, and they can be composed with other models.

In this paper we focus on model transformation testing and model composition and demonstrate how both tasks can benefit, in different ways, from the automation of the fundamental task of *model comparison*. Then, we provide a definition and derive requirements for model comparison incrementally and propose a rule-based approach, with tool support, for comparing models of arbitrary languages and technologies. Finally, we discuss how the results of model comparison can be utilized for model transformation and composition/merging[1].

GaMMa'06, May 22, 2006, Shanghai, China.

2. MOTIVATION AND BACKGROUND

In this section we present the motivation for our approach to model comparison, as well as relevant work on the subject.

2.1 Motivation

Our primary motivation to consider model comparison stemmed from our need to define a generic model merging language: the Epsilon Merging Language (EML), which is part of the Extensible Platform for Specification of Integrated Languages for mOdel maNagement (EPSILON) [1], a platform of model-oriented languages partly developed in the context of the EU integrated MODELWARE project[2]. However, during the development of EML we determined that model comparison is important enough to constitute a stand-alone operation from which other aspects of model driven development (such as model transformation testing) can also benefit.

2.2 Background and Related Work

A large number of techniques and algorithms have been proposed for document comparison with the majority of them applying to textual files [3, 4] or structured documents [5, 6] (e.g. XML files). However, as discussed in [7], although models can eventually be serialized in structured text files, file and tree-based approaches are unsuitable for comparing models since they operate at a significantly low abstraction level.

The majority of approaches that specifically target model comparison generally apply to a single modelling language such as UML as discussed in [7]. Nevertheless, in the context of Model Driven Development and with the advent of approaches based on Domain Specific Languages [8, 9], it is essential that models of different languages and technologies can be compared as well.

In the context of metamodel agnostic approaches, in [10] model comparison is performed using MOF unique identifiers. This approach exploits the fact that MOF compliant

[1]In this paper we use the terms *composition* and *merging* interchangeably

model repositories assign a unique and persistent identifier to each model element upon creation. Therefore, when comparing different versions of the same model, elements can be matched according to their unique identifiers. Still, the applicability of this approach is limited to MOF models that are descendant versions of a common ancestor model.

A metamodel agnostic approach for comparing models of diverse technologies is the Xlinkit approach proposed in [11]. Xlinkit is a general-purpose tooklit for checking consistency of XML documents. In [12], Xlinkit is used to check the consistency of MOF-based models exploiting the fact that such models are stored as XMI (a dialect of XML) documents. Since other modelling technologies such as EMF [13] and the Microsoft DSL Framework [8] also advocate model serialization in XML, XLinkit can be even used to compare models of different modelling technologies. Even so, the concrete syntax of the expression language XLinkit employs is XML-based and this results in lengthy and difficult to read and maintain specifications. Moreover, we regard the storage level (XML) as an inappropriately low abstraction level for expressing constraints on complex models since there are certain model-specific features such as inheritance and details of cross-referencing, the semantics of which are defined in the metamodel rather than in the serialized XML documents.

3. MODEL COMPARISON, COMPOSITION AND TRANSFORMATION TESTING

In this section we discuss the importance of model comparison for the tasks of model composition and model transformation testing.

3.1 Model Composition

In the relevant literature, the subject of composition of structured artefacts such as models [14, 15], database schemas [16, 17, 18] and XML schemas [19] has been extensively studied. As discussed in all of the above, a prerequisite of composition is the identification of common elements contained in the two sources so that the merged artefact does not contain duplicated information. Moreover, to ensure that the result will not contain contradictions, it is vital to check that the source artefacts conform to each other; i.e., they are composable.

3.2 Model Transformation Testing

While model transformation is a key concept of Model Driven Development [20] and a number of transformation languages have been proposed [21, 22], the aspect of transformation testing is significantly underdeveloped. As discussed in [23], without automated testing it is difficult, especially for large models, to check whether a transformation is complete and has the intended functionality. Therefore, the authors identify the need for a generic model comparison mechanism as a fundamental prerequisite for a transformation testing framework. In further work [24], the authors suggest manual construction of the expected outcome of the transformation and comparison with the actual outcome of the transformations using a simple graph-comparison algorithm since the models in comparison (expected and actual) are of the same metamodel. While this makes model transformation testing feasible, our view is that constructing the expected outcome manually is not an efficient and scalable approach.

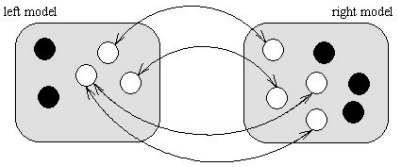

Figure 1: Initial classification

4. REQUIREMENTS FOR MODEL COMPARISON

In this section we discuss the problem of comparing two models of arbitrary metamodels and even of diverse modelling platforms (e.g. MOF [25], EMF [13], GME [26] XML or Microsoft's Domain Specific Languages initiative [8]) at a high level of abstraction.

Initially, we have to define what we wish to achieve when comparing two models, and we develop an answer to this question in an incremental manner. Thus, as an initial requirement, a comparison algorithm should examine two input models ($left$ and $right$) and partition their elements into the following categories:

1. Elements for which matching elements exist in the opposite model
2. Elements for which matching elements do not exist in the opposite model

In Figure 1, white nodes represent the elements of the first category while black nodes represent elements of the second category. Discussion on how matching is performed is not appropriate at the current level of abstraction and is deferred until later.

The following concrete example demonstrates that the classification could be more fine grained: Consider that both the left and the right model are UML models. Informally, a pair of UML classes with the same name under matching packages could be assumed to form a match. However, what happens if the class of the left model is declared as abstract while the class of the right model is declared as concrete? While the pair of classes may still be considered to be a match (since they likely represent the same conceptual artefact), there is a *conformance* mismatch between them. This example raises the issue of conformance between matching elements. So, the elements of category 1 can be further classified into two distinct sub categories.

1. Elements for which matching elements exist in the opposite model
   (a) Elements that conform to their matching elements in the opposite model
   (b) Elements that do not conform to some matching elements in the opposite model

As with matching, the details of how to decide whether two elements conform to each other is deferred until the next section. The refined classification is represented graphically in Figure 2. There, grey nodes connected with dotted lines represent elements of category 1b, while white nodes represent elements of category 1a.

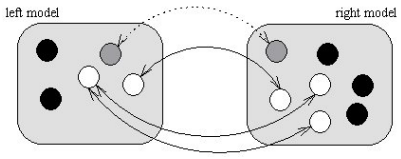

Figure 2: Refined classification

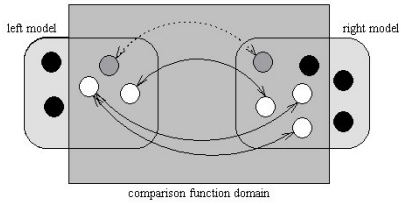

Figure 3: Further refined classification

Category 2 can also be refined into more specialized subcategories depending on whether the elements belonging to it are included or not in the domain of the comparison operation. In the case of elements that are included in the domain, the comparison operation has determined that there are no matching elements for them in the opposite model. On the other hand, existence of elements that are excluded from the domain of the operation may indicate that the comparison operation is incomplete or that it intentionally ignores them.

2. Elements for which matching elements do not exist in the opposite model
   (a) Elements that are included in the domain of the comparison operation
   (b) Elements that are excluded from the domain of the comparison operation

Schematically this is represented in Figure 3. There, the grey rectangular surface represents the domain of the comparison operation. Therefore, black nodes that exist inside it represent elements of category 2*a* while black circles outside it represent elements of category 2*b*.

Throughout this section, we have progressively derived a precise definition that model comparison is a function that operates on two models and partitions their elements into the four categories discussed above. In the sequel, we discuss on the usefulness of this classification and present a modular and comprehensive automated approach for achieving it in practice.

5. INTERPRETATION OF THE RESULT OF MODEL COMPARISON

As discussed in Section 4 the result of model comparison is a classification of the elements of the compared models into four distinct categories. In this section we discuss how those results can be interpreted from the perspectives of model transformation testing and model composition. The purpose of this section is not to provide an exhaustive list of possible interpretations for all different cases and modes of transformations and compositions, but rather to give examples that justify that both of these tasks benefit from this classification.

5.1 Interpretation for Transformation Testing

As with all forms of testing, there is always the possibility that the artefact under testing is correct and the testing mechanism (in our case transformation and comparison) is faulty. In this case, testing may not detect existing faults or even detect non-existing faults in the tested artefact. However, since there is no generally accepted solution to this problem we generally assume that the testing mechanism is correct and any identified issues are a result of the artefact under test.

Under this assumption, elements of category 1*a* (matching and conforming) indicate that the transformation has correctly mapped them from the source to the target model.

Elements of category 1*b* (matching but not conforming) suggest that the transformation may be *incorrect* since some features of the source elements may have not been properly mapped to features of the target elements.

Elements of category 2*a* (not matching but belonging to the domain of the comparison) which belong to the source model hint that the transformation may be *incomplete* (intentionally or unintentionally) since it has not mapped those elements into respective elements in the target model.

Elements of category 2*a* that belong to the target model are most probably a result of an erroneous transformation.

Finally, elements belonging to category 2*b* (not matching and not belonging to the domain of the comparison) may indicate that the comparison is *incomplete*.

5.2 Interpretation for Model Composition

For model composition, possible interpretations of the result of comparison are significantly different than those discussed for model transformation testing. More specifically:

Elements of category 1*a* should be merged into a single element in the composed model.

Elements of category 1*b* indicate that there are conformance issues in the model that need to be resolved (either automatically or manually) before the models can be composed.

For elements of category 2*a* there is no generic interpretation and they are treated differently according to the specific requirements of the composition. For example when merging two partially overlapping model, one may consider transforming the elements of this category as-is to the target model. In the case of Y development [27], where a platform independent model and a platform description model are composed, not detecting matching platform specific concepts for some platform independent concepts is a vital issue that makes the two models incomposable.

Finally, elements of category 2*b* indicate that the comparison may be incomplete.

6. RULE-BASED MODEL COMPARISON

Having defined the requirements and benefits of model comparison, in this section we present a concrete metamodel-agnostic and technology-independent rule-based comparison language we implemented in the context of the Epsilon Merging Language (EML). In the sequel, we discuss the abstract and concrete syntax of the language as well as details about the rules' execution scheduling.

6.1 Rule Syntax

The abstract and concrete syntaxes of a match rule are outlined in Figure 4 and Listing 1; examples will be pre-

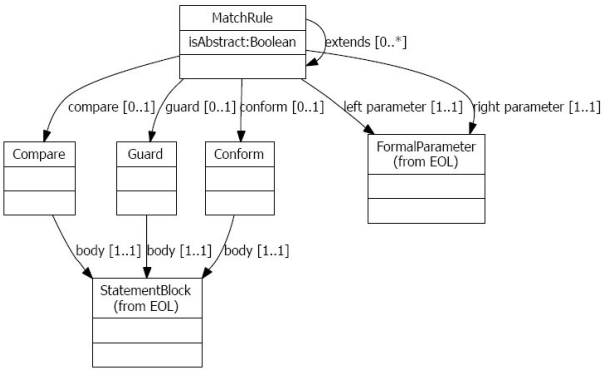

Figure 4: Match Rule Abstract Syntax

sented in the sequel.

Listing 1: Match Rule Concrete Syntax

```
(abstract)? rule <name>
 match <left-name> : <left-type>
 with <right-name> : <right-type>
 (extends <ruleName>*)? {

 (guard {
   <body>
 })?

 (compare {
   <body>
 })?

 (conform {
   <body>
 })?
}
```

Each match rule has a unique name. It also defines a left and a right parameter that declare the types of the elements it applies on as well as the rules it inherits (extends). The semantics of rule inheritance are discussed in the sequel. A rule also contains three optional parts; *guard*, *compare* and *conform*.

The *guard* part is used to further restrict the domain of the rule beyond the default checking of the elements' types against the declared types of the parameters. To achieve this, it typically performs queries on features of the elements against which it is evaluated and returns a *boolean* value that shows whether the rule applies to them.

If the *guard* returns *true*, then the *compare* part of the rule is executed. The role of the *compare* part is to decide if the elements that are compared form a match. To decide that, usually only a minimal set of features of the elements need to be examined. For example, in a rule that compares UML classes, it is usually enough to check that they have the same name and belong to matching packages (the decision of whether the packages match should be deferred to another rule) to decide that they form a match.

If the *compare* parts return *true*, the elements under comparison form a match (Category 1). In the sequel, the *conform* part of the rule is executed to further categorize the pair of elements as belonging to sub-category 1*a* or 1*b* (*conforming* or *not-conforming*). In this part, more properties of the matching elements are typically checked. For example, the user can define that matching UML attributes (e.g. attributes with the same name that belong to matching classes) must be of the same type as well.

With regard to the semantics of rule inheritance, for every rule that extends one or more other rules, in addition to its *guard*, *compare* and *conform* parts, all the respective parts of its ancestor rules are also executed and they must all return *true* if the rule is to continue with the next part.

6.2 Rule Execution Scheduling

Non-abstract match rules are executed sequentially in the order that they are declared[2]. Each rule is executed for all the instances of the $left$ and $right$ parameters and the results are stored in an internal *cache* to avoid re-calculation.

An exception to the sequential execution of the rules happens when the body of a *guard*, *compare* or *conform* part invokes the built-in *matches*(*element*) method. The *matches*() operation, checks the *cache* to find out if the specified pair of elements (source and parameter) are known to form a match and returns *true* or $false$ respectively. In the case the pair is not in the trace (it has not been checked yet), it tries to find and execute suitable match rules that can decide if the elements form a match. The final product of this process is a populated match trace that contains information about each pair of elements that have been compared by the match rules. This product can be stored and/or visualised as discussed later.

6.3 Expression language

In the Epsilon Merging Language, we use the Epsilon Object Language (EOL) [28] for defining the functionality of the bodies of the rules' *guard*, *compare* and *conform* parts. EOL is a generic model navigation and management language that extends the navigational features of the Object Constraint Language (OCL) [29] with useful features such as multiple model access, statement sequencing, conventional programming constructs and model modification capabilities. The purpose of EOL is to form a reusable core language that can be embedded in task-specific model management languages across the Epsilon platform.

A basic requirement of the architecture of EOL is that it must be independent of the underlying technology with which the models it manages are defined (e.g. MOF, EMF, MSDSL, GME). Thus, EOL and the languages in which it is embedded can manage models of diverse technological platforms. For example, the Epsilon Merging Language, the comparison part of which is demonstrated in this paper, can compare and merge a MOF with an EMF model. More details about the rationale behind EOL and Epsilon are discussed in [28].

In the sequel, we present concrete examples of rules to clarify the approach described above.

7. CASE STUDY

In this case study we use the presented comparison part of EML to compare a UML 1.4 model against a relational

[2]the order of execution does not actually affect the comparison result

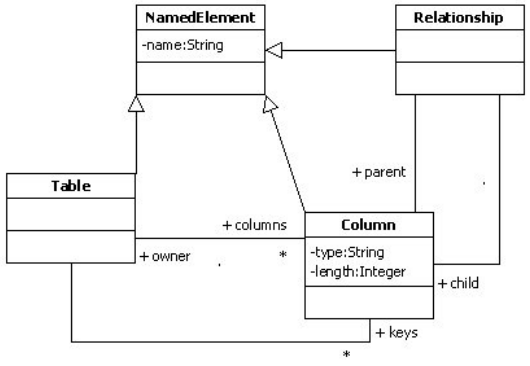

Figure 5: Relational metamodel

database model that conforms to the metamodel displayed in Figure 5. In our scenario, the database model has been automatically derived from the UML model using a transformation (e.g. an ATL [22] or QVT [21] transformation) and we wish to test whether the transformation has produced the intended result.

7.1 Transformation Functionality

In this section we discuss the intended functionality of the *UML → Database* transformation. Although, the functionality could be represented in terms of the concrete syntax of a transformation language from those discussed in the prequel, we anticipate that outlining its functionality will provide better understanding to the reader.

Class → Table, Key For each Class of the UML model, a Table with the same name is produced in the Database model. Moreover, a key named *id* is added to the table.

Attribute → Column For each Attribute of each Class in the UML model, a Column with the same name and an appropriate type is added to the respective Table

Many-To-Many Association → Intermediate Table, Relationships, Keys For each many-to many Association in the UML model, an intermediate Table is produced in the Database Model that has the concatenation of the names of the ends of the association as name. The table contains a double-key consisting of two columns named after the ends of the association adding the suffix *id*. Moreover, two relationships linking the keys of the intermediate table with the keys of the primary tables are created.

One-To-Many Association → Relationship, Column For each one-to-many Association in the UML model, a new column is added in the end of the association that has its multiplicity set to 1 and a relationship between it and the *id* column of the opposite table is created.

7.2 Comparison Functionality

Having defined the requirements for the UML to Database transformation, in Listing 2, we define the match rules that can test if an implementation of the transformation satisfies them.

Listing 2: Case-study match rules

```
-- (1) ModelElement2NamedRule
abstract rule ModelElement2NamedElement
  match me : UML!ModelElement
  with ne : Database!NamedElement {

  compare {
    return me.name = ne.name;
  }
}

-- (2) Class2Table
rule Class2Table
  match cls : UML!Class
  with tbl : Database!Table
  extends ModelElement2NamedElement {

}

-- (3) Class2Key
rule Class2Key
  match cls : UML!Class
  with col : Database!Column {

  compare {
    return col.name = 'id' and
           cls.matches(col.owner);
  }

  conform {
    return col.owner.keys.includes(col);
  }

}

-- (4) Attribute2Column
rule Attribute2Column
  match att : UML!Attribute
  with col : Database!Column
  extends ModelElement2NamedElement {

  compare {
    return att.owner.matches(col.owner);
  }

  conform {
    return att.type.matches(col.type);
  }

}


-- (5) ManyToManyAssociation2Table
rule ManyToManyAssociation2Table
  match assoc : UML!Association
  with t : Database!Table {

  guard {
    return assoc.connection.
      forAll(ae|ae.isMultiple());
  }

  compare {
    def ends : Sequence(UML!AssociationEnd);
    def name1 : String;
    def name2 : String;
    ends := assoc.connection.asSequence();
    name1 := ends.at(0).name.ftuc();
    name2 := ends.at(1).name.ftuc();

    return (t.name = name1 + name2) or
      (t.name = name2 + name1);
  }

}

-- (6) OneToManyAssociationEnd2Relationsip
rule OneToManyAssociationEnd2Relationship
  match ae : UML!AssociationEnd
  with rel : Database!Relationship {

  guard {
    return ae.isMultiple() and
     ae.getOpposite().isMultiple() = false;
  }

  compare {
    return ae.participant.matches(rel.child.owner)
    and ae.getOpposite().participant.
        matches(rel.parent.owner);
  }
}

-- (7) OneToManyAssociationEnd2Column
rule OneToManyAssociationEnd2Column
  match ae : UML!AssociationEnd
  with col : Database!Column {

  guard {
    return ae.isMultiple() and
     (not ae.getOpposite().isMultiple());
  }

  compare {
    return ae.participant.matches(col.owner)
    and col.name = ae.getOpposite().name + 'Id';
  }
}

-- (8) ManyToManyAssociationEnd2Relationship
rule ManyToManyAssociationEnd2Relationship
  match ae : UML!AssociationEnd
  with rel : Database!Relationship {

  guard {
```

```
    return ae.isMultiple() and
     ae.getOpposite().isMultiple();
  }

  compare {
    return
      ae.association.matches(rel.child.owner) and
      ae.participant.matches(rel.parent.owner);
  }

}

-- (9) ManyToManyAssociationEnd2Key
rule ManyToManyAssociationEnd2Key
  match ae : UML!AssociationEnd
  with col : Database!Column {

  guard {
    return ae.isMultiple()
    and ae.getOpposite().isMultiple();
  }

  compare {
    return col.owner.matches(ae.association)
    and col.name = ae.name + 'Id';
  }

  conform {
        return col.owner.keys.includes(col);
  }
}

-- (10) Primitive2DataType
rule Primitive2DataType
  match cls : UML!Class
  with dt : Database!DataType {

  guard {
    return cls.stereotype.
      exists(st|st.name = 'primitive');
  }

  compare {

  def c : String;
  def d : String;
  c := cls.name;
  d := dt.name;

  return (c = 'String' and d = 'TEXT') or
    (c = 'Integer' and d = 'INTEGER') or
    (c = 'Boolean' and d = 'BIT');
  }
}

-- Helper operations

operation UML!AssociationEnd isMultiple() : Boolean {
  def upper : Integer;
  upper := self.multiplicity.range.
    asSequence().first().upper;
  return (upper > 1) or (upper = -1);
}

operation UML!AssociationEnd getOpposite()
  : UML!AssociationEnd {
  return self.association.connection.
    reject(ae|ae = self).first();
}
```

7.3 Discussion on the Match Rules

In this section we briefly discuss the functionality of the match rules presented in Listing 2 in the order that they appear.

(1) ModelElement2NamedElement compares a UML ModelElement with a Database NamedElement by checking that their names match. As this is an abstract rule, it is executed only as a part of rules that extend it.

(2) Class2Table compares a UML Class with a Database Table. Since the rule does not define a compare part, only the compare part of *ModelElement2Named* will be executed thus checking only the that the Class and the Table have matching names.

(3) Class2Key compares a UML Class with a Database Column. To match, the name of the column must be *'id'* and its owning Table must match with the Class (compared in Rule 2).

(4) Attribute2Column compares a UML Attribute with a Database Column. To match, the elements must have the same name (since the rule extends Rule 1) and belong to matching owners (compared in Rule 2). Moreover, to conform, their types must match (compared in Rule 8).

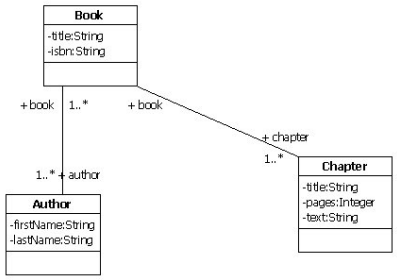

Figure 6: UML 1.4 model

(5) ManyToManyAssociation2Table compares a UML Association which has *multiple* AssociationEnds[3] with a Database (intermediate) Table. To match, the name of the table must be the concatenation of the names of the ends of the association in any order.

(6) OneToManyAssociationEnd2Relationship compares a *multiple* UML AssociationEnd (the opposite AssociationEnd of which must be *single*) with a Database Relationship. To match, the type of the *multiple* end must match the owner of the *child* column (compared in Rule 2) and the type of the *single* end must match the owner of the *parent* column (compared in Rule 2) of the relationship.

(7) OneToManyAssociationEnd2Column compares a multiple UML AssociationEnd (the opposite AssociaionEnd of which must be single) with a Relational Column. To match, the type of the AssociationEnd must match the Table that owns the Column (compared in Rule 2) and the name of the Column must be the name of the opposite AssociationEnd suffixed with the string 'id'.

(8) ManyToManyAssociationEnd2Relationship compares a multiple UML AssociationEnd (the opposite AssociationEnd of which must be also multiple) with a Database Relationship. To match, the owning Association of the AssociationEnd must match the owner Table of the child key of the Relationship (compared in Rule 5) and the type of the AssociationEnd must match the owner Table of the parent key of the Relationship (compared in Rule 2).

(9) ManyToManyAssociationEnd2Key compares a multiple UML AssociationEnd (the opposite AssociationEnd of which must be also multiple) with a Database Column. To match, the owner Table of the Column must match the owning Association of the AssociationEnd and the name of the column must be the name of the AssociationEnd suffixed with a the string 'id'. To conform, the Column should be a key of its owning Table.

(10) PrimitiveToDataType compares a UML class that is stereotyped as *primitive* with a Database DataType. To match, they must have matching names.

In the sequel, we execute the rules on the sample models displayed in Figures 6 and 7. The result of the comparison of the two models is visualized in a user interface that displays both models as hierarchical structures (trees) where each model element is marked with a different icon according to the category it belongs to (matching, non-matching, conforming, not-conforming, not matched). and selection of an element in a model triggers automatic selection of its

[3]with the terms *multiple* and *single* we refer to AssociationEnds the multiplicity of which is greater than one and exactly one respectively

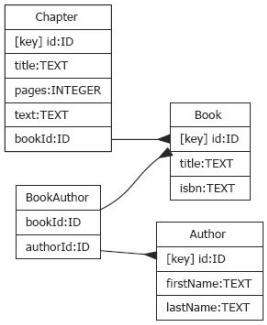

Figure 7: Database model

matching elements in the opposite model. In the context of our example, while AssociationEnds *author* and *book* match the columns $authorId$ and $bookId$ respectively, they do not conform to them since the columns are not declared to be keys of the table $AuthorBook$ as requested in Rule 9. This means that the implementation of our transformation that has produced the Database model is faulty and needs correction.

8. CONCLUSIONS AND FURTHER WORK

In this paper we have discussed the requirements for model comparison as well as its usefulness for model management tasks such as model composition and model transformation testing. We have presented a rule-based approach for performing automated comparison on diverse models and demonstrated an example of the concrete syntax and the functionality of our approach.

While model comparison was considered to be a part of the model composition process, our research has shown that it is a standalone task from which other model management tasks can benefit as well. Therefore in the future we plan to separate model comparison functionality from our merging language (EML) and provide it in a standalone language.

An issue about developing a standalone model comparison language is that the results of the comparison must be stored in a way that makes them reusable for further processing. For this reason we will consider alignment with the ModelWeaver framework [30], a generic framework for defining relationships between elements of different models. To achieve alignment we shall consider extending the Core Weaving Metamodel with relationships and elements specific to model comparison.

Moreover, we plan to enhance the functionality of the match rules by introducing the concepts of unique persistent identities. In popular modelling technologies such as MOF and EMF, each element is assigned a unique identifier upon creation. This feature is particularly useful when comparing models that are descendants of a common ancestor (e.g. different versions of a common model). In this case, models are expected to contain a significant number of elements with matching unique ids which could be identified as matching *a priori*. Another possible extension to our approach is to use dictionaries of synonyms, a technique widely used in database schema merging approaches [16], to avoid hard-coding synonyms in match rules as we did for example in Rule 10 of the case study.

9. ACKNOWLEDGEMENTS

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.

10. REFERENCES

[1] Dimitrios S. Kolovos. Extensible Platform for Specification of Integrated Languages for mOdel maNagement (Epsilon), Official Web-Site. http://www.cs.york.ac.uk/~dkolovos/epsilon/.
[2] Modelware IST Project. http:///www.modelware-ist.org.
[3] Eugene W. Myers. An O(ND) Difference Algorithm and Its Variations. *Algorithmica*, 1(2):251–266, 1986.
[4] Paul Heckel. A technique for isolating differences between files. *Commun. ACM*, 21(4):264–268, 1978.
[5] Amélie Marian and Serge Abiteboul and Grégory Cobéna and Laurent Mignet. Change-Centric Management of Versions in an XML Warehouse. In *The VLDB Journal*, pages 581–590, 2001.
[6] Amelie Marian. Detecting Changes in XML Documents. In *ICDE '02: Proceedings of the 18th International Conference on Data Engineering (ICDE'02)*, Washington, DC, USA, 2002.
[7] Dirk Ohst, Michael Welle and Udo Kelter. Differences between Versions of UML Diagrams. In *9th European Software Engineering Conference*, pages 227–236. ACM Press, 2003.
[8] Microsoft Domain Specific Languages Framework, Official Web-Site. http://msdn.microsoft.com/ vstudio/teamsystem/ workshop/DSLTools/default.aspx.
[9] MetaCase. Meta-Edit+. http://www.metacase.com.
[10] Marcus Alanen and Ivan Porres. Difference and Union of Models. Technical Report 527, TUCS, April 2003.
[11] Christian Nentwich, Licia Capra, Wolfgang Emmerich and Anthony Finkelstein. xlinkit: A Consistency Checking and Smart Link Generation Service. *ACM Transactions on Internet Technology*, 2(2):151–185, May 2002.
[12] Christian Nentwich, Wolfgang Emmerich, Anthony Finkelstein and Erns Ellmer. Flexible Consistency Checking. *ACM Transactions on Software Engineering and Methodology*, 12(1):28–63, 2003.
[13] Eclipse.org. Eclipse Modelling Framework. http://www.eclipse.org/emf/.
[14] Siobhan Clarke. Extending UML Metamodel for Design Composition. In *Workshop on Multi-Dimensional Separation of Concerns in Software Engineering (ICSE 2000)*, 2000.
[15] Harald Kuhn, Franz Bayer, Stefan Junginger and Dimitris Karagiannis. Enterprise Model Integration. In Bauknecht, K.; Tjoa, A M.; Quirchmayr, G., editor, *Proceedings of the 4th International Conference EC-Web 2003*, pages 379–392. Springer Verlag, September 2003.
[16] C. Batini, M. Lenzerini, S.B. Navathe. A Comparative Analysis of Methodologies for Database Schema Integration. *ACM Computing Surveys*, 18(4):323–364, December 1986.
[17] P. Buneman, S.Davidson and A.Kosky. Theoretical Aspects of Schema Merging, 1994.
[18] S.B. Yao, V. Waddle, and B. Housel. View modeling and integration using the functional data model. *IEEE Transactions in Software Engineering*, 8(6):544–553, 1982.
[19] Ralf Behrens. A Grammar Based Model for XML Schema Integration. In *BNCOD 17: Proceedings of the 17th British National Conferenc on Databases*, pages 172–190, 2000.
[20] Shane Sendall and Wojtek Kozaczynski. Model Transformation the Heart and Soul of Model-Driven Software Development . *IEEE Software*, 20(5):42–45, September/October 2003.
[21] QVT Partners Official Web-Site. http://qvtp.org/.
[22] Atlas Transformation Language, official web-site. http://www.sciences.univ-nantes.fr/lina/atl/.
[23] Yuehua Lin, Jing Zhang, and Jeff Gray. Model Comparison: A Key Challenge for Transformation Testing and Version Control in Model Driven Software Development. In *Object Oriented Programing, Systems, Languages and Applications*, 2004.
[24] Yuehua Lin, Jing Zhang, and Jeff Gray. A Testing Framework for Model Transformations. http://www.gray-area.org/Pubs/transformation-testing.pdf.
[25] Object Management Group. Meta Object Facility (MOF) 2.0 Core Specification. http://www.omg.org/cgi-bin/doc?ptc/03-10-04.
[26] Generic Modeling Environment. http://www.isis.vanderbilt.edu/Projects/gme/.
[27] Jean Bezivin. On the Unification Power of Models. *Software and System Modeling (SoSym)*, 4(2):171–188, 2005.
[28] Dimitrios S. Kolovos, Richard F.Paige and Fiona A.C. Polack. The Epsilon Object Language. Under review, 2006. http://www-users.cs.york.ac.uk/~dkolovos/publications/eol.pdf.
[29] Object Management Group. UML 2.0 OCL Specification. http://www.omg.org/docs/ptc/03-10-14.pdf.
[30] Marcos Didonet Del Fabro, Jean Bezivin, Frederic Jouault, Erwan Breton, Guillaume Gueltas. AMW: A Generic Model Weaver. *Premieres Journees sur l'Intgenierie Dirigee par les Modeles*, 2005.