Semi-automatic Model Integration using Matching Transformations and Weaving Models

Marcos Didonet Del Fabro Patrick Valduriez
ATLAS Group, INRIA & LINA
University of Nantes
+33 (0)2 51 12 58 08
marcos.didonet-del-fabro@univ-nantes.fr, patrick.valduriez@inria.fr

ABSTRACT
Model transformations are at the heart of model driven engineering (MDE) and can be used in many different application scenarios. For instance, model transformations are used to integrate very large models. As a consequence, they are becoming more and more complex. However, these transformations are still developed manually. Several code patterns are implemented repetitively, increasing the probability of programming errors and reducing code reusability. There is not yet a complete solution that automates the development of model transformations. In this paper we propose a novel approach that uses matching transformations and weaving models to semi-automate the development of transformations. Matching transformations are a special kind of transformations that implement heuristics and algorithms to create weaving models. Weaving models are models that capture different kinds of relationships between models. Our solution enables to rapidly implement and to customize these heuristics. We combine different heuristics, and we propose a new metamodel-based heuristic that exploits metamodel data to automatically produce weaving models. The weaving models are derived into model integration transformations.

Categories and Subject Descriptors
D.2.11 [**Software architectures**]: Domain-specific architectures. D.2.12 [**Interoperability**]: Data mapping.

General Terms
Algorithms, Standardization, Languages

Keywords
Model engineering, matching transformations, model weaving.

1. INTRODUCTION
Model transformations are a central component in model driven engineering practices. There are many model transformation languages emerging from industrial and academic efforts [3][12][14][15]. As a consequence, there are an increasing number of model transformations that are being developed for different applications scenarios. For instance, there are transformations to provide tool interoperability, to translate from textual to graphical representations, or to merge models.

However, the development of transformations involves many repetitive tasks. Consider for example a generic model integration scenario that transforms one source model into one target model. The transformation development consists of creating rules that transform a set of elements of the source model into a set of elements of the target model. The properties of these elements are transformed using a set of transformation expressions. Most part of these expressions consists of 1-to-1 relationships or other common patterns, such as nesting or concatenation of elements.

These transformations are often created manually. To the best of our knowledge, there is not a MDE approach that provides enough generic mechanisms to semi-automate the development of transformations. A semi-automatic process based on well-defined patterns brings many advantages: it accelerates the development time of transformations; it diminishes the errors that may occur in manual coding; it increases the quality of transformational code.

The discovery of transformation patterns to integrate models is closely related to schema and to ontology matching approaches (see the survey at [22]). These approaches aim at discovering semantic relationships between elements of different schemas or ontologies. These relationships are used for different purposes, such as ontology alignment [9][19] or data translation [6]. However, these approaches have some drawbacks. Most part of solutions cannot be applied to models conforming to different metamodels. Metamodels are models that describe the structure of models. The distance between the conceptual basis (models) and the implementation (heuristics) is too important. This makes difficult to decompose and to customize different heuristics. There is no support for different kinds of relationships between models. Hence, native constructs of transformation languages are not supported, such as rule inheritance or nested relationships.

In this paper, we present a novel solution to semi-automate the development of model transformations. We propose the execution of matching transformations. Matching transformations are transformations that select a set of elements from a set of input models and that produce links between these elements. These links are captured by a weaving model, as we proposed in [7]. The weaving model conforms to extensions of a weaving metamodel. We define links that act as specifications for model integration transformations. Model integration transformations are used in standard model integration applications.

Matching transformations enable to rapidly implement new or to adapt heuristics to create weaving models. In addition, we propose a new metamodel-based heuristic that exploits the

SAC'07, March 11-15, 2007, Seoul, Korea.

internal features of the set of input metamodels to produce weaving models. This heuristic is executed together with a link rewriting method that analyzes the weaving metamodel extensions to produce frequently used transformation patterns.

The main contributions of this paper are the following. We propose a solution to semi-automate the development of model transformations. We innovate by using matching transformations to allow an easy development of different matching heuristics or algorithms. We propose a metamodel-based heuristic that exploits the information from the set of input metamodels and from the weaving metamodel. These matching transformations automatically create weaving models. The weaving models are derived into model integration transformations.

This paper is organized as follows. Section 2 is the motivating example. Section 3 presents the general architecture. Section 4 presents weaving metamodel extensions that capture different kinds of relationships between models. Section 5 describes the matching transformations in more details. Section 6 presents how to derive a weaving model into executable model integration transformations. Section 7 presents a general discussion. Section 8 presents the related work. Section 9 concludes.

2. MOTIVATING EXAMPLE

We motivate the necessity to automatically create model transformations using two simple metamodels *MM1* and *MM2*. Both metamodels are illustrated in Figure 1. They describe the teachers and the students of different educational institutions. These metamodels have similar attributes and references, but they are organized differently. Metamodel *MM1* contains an abstract class *Person*, with attributes *name*, *SSN (Social Security Number)*, *street*, *city* and *zip_code*. The class *Teacher* inherits from *Person*, and it has the *affiliation* of the teacher. *MM1* has two types of students: undergraduate students (*Undergraduate*) and master students (*Master*). Only master students have an *advisor*. Metamodel *MM2* does not support inheritance. *MM2* contains a class *Professor* and only one class *Student*. The presence of an advisor indicates if the student is undergraduate or master. The address of the professors and the students is factored out on the class *Address*.

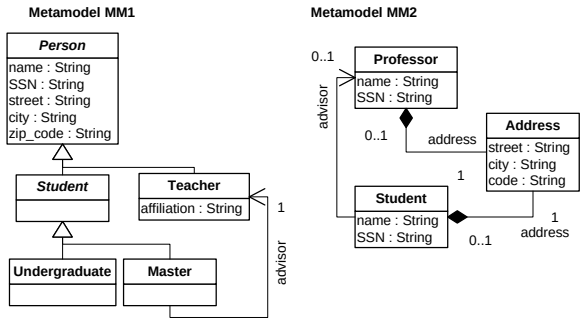

Figure 1. Two simple metamodels

In Figure 2 we show a model transformation used to transform models conforming to *MM1* (i.e., source model) into models conforming to *MM2* (i.e., target model). The transformation is written in ATL (a complete description of the language is available in [15]). We choose ATL because it provides a simple syntax adapted for model transformations.

This transformation has 3 rules; each rule matches one element of the source model and creates elements in the target model. The transformation developer must know that *Teacher* is transformed into *Professor* and that *Master* and *Undergraduate* are transformed into *Student*. After that, all the attributes and references of each class must be translated as well (*name, SSN, address, advisor, street, code,* etc.).

```
rule CreateProfessor {
  from source : MM1!Teacher
  to target : MM2!Professor (
     name <- source.name,
     SSN <- source.SSN,
     address <- address ),
  address : MM2!Address (
     street <- source.street,
     city <- source.city,
     code <- source.zip_code )
}

rule CreateStudent1 {
  from source : MM1!Undergraduate
  to target : MM2!Student (
  -- copy bindings from CreateProfessor
    )
}
rule CreateStudent2 {
  from source : MM1!Master
  to target : MM2!Student (
    advisor <- source.advisor
  -- copy bindings from CreateProfessor
    )
}
```

Figure 2. Transformation definition

This transformation has basically two kinds of expressions: transformations between self contained elements (i.e., classes), and the setup of their properties (i.e., attributes and references). Thus, in the three rules, the transformation has a source class and a set of target classes. The rule *CreateProfessor* assigns the attributes of *Teacher* to *Professor*. These attributes are inherited from *Person*. The attributes from both classes have similar properties, such as name and type. These attributes are transformed in the containing class, or in a newly created class (*Address*). The same set of expressions must be rewritten in *CreateStudent1* and in *CreateStudent2* rules, because *Undergraduate* and *Master* inherit from *Student*, that inherits from *Person*. The transformation developer has two choices: to copy and paste the code, or to apply rule inheritance predicates.

These expressions are common patterns in transformations that involve similar metamodels, for example in model integration or in model evolution scenarios. These transformations can be very large depending on the source and target metamodels. The automatic discovery of these transformation patterns can increase the development speed of model transformations. The intervention of qualified transformation developers is left essentially to more complex expressions that do not occur frequently and that cannot be created automatically.

In order to automate the development of transformations, it is necessary to discover the different kinds of relationships (links) between metamodel (or model) elements. These links must be saved in another model. This model can be validated or modified by the transformation developer.

Heuristics similar to ontology and schema matching can be used to discover these links. However, model transformations can be executed over several different source and target metamodels, with different attributes, relations, properties, etc. The patterns applied vary from case to case. Consequently, it is also important to have efficient ways to implement new heuristics and to adapt existing heuristics.

As final step, these links must be translated into the correct transformation expressions, for instance links between attributes of abstract classes must be translated into bindings (a binding is denoted by the "←" symbol) in the inherited classes.

3. GENERAL ARCHITECTURE

This section presents an overview of the architecture to semi-automate the production of model transformations. The architecture is illustrated in Figure 3. It is formed by three main components: the weaving engine, the transformation engine and the matching management component.

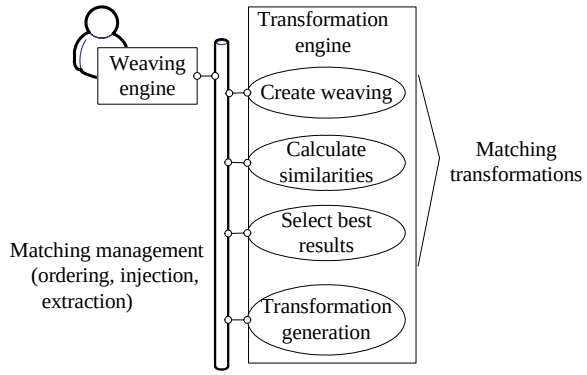

Figure 3. General architecture

3.1 Weaving Engine

The weaving engine supports the specification of the different transformation patterns. The weaving engine is the only component that enables manual user input. It provides interfaces to create/update weaving metamodels and weaving models.

The pattern definitions are encoded as typed links in a weaving metamodel. For instance, the link between the *SSN-SSN* attributes are typed as an *AttributeEqual* link, indicating the equality between two attributes (we describe these links further in this paper). The weaving models capture the different kinds of links defined in the weaving metamodel (a formal definition of weaving models and metamodels can be found at [7]).

The weaving engine exchanges the weaving models and metamodels with the transformation engine through the *matching management* component.

3.2 Transformation Engine

The transformation engine executes different kinds of model transformations. The process is divided in two phases: the *matching* phase, and the *transformation generation* phase. We explain them below.

3.2.1 Matching

The matching phase discovers the relationships between a set of input models and creates a weaving model. The whole process is encapsulated in a model management operation called *Match* [4]. The *Match* operation takes two models M_a and M_b as input and produces a weaving model M_w as output. M_a and M_b conform to MM_a and MM_b; M_w conforms to MM_w.

$$M_w : MM_w = Match\ (M_a : MM_a, M_b : MM_b).$$

The *Match* operation is semi-automatic, i.e., it is an interactive process that alternates between the automatic execution of matching transformations and the manual refinement of weaving models in the weaving engine. Matching transformations are transformations that execute different heuristics to produce a weaving model.

There are three kinds of matching transformations. The first kind creates a weaving model with links between the elements of the input models. However, it is not possible to create a weaving model with only correct links between the model elements in a single transformation. For instance, we create links between *name-name* attributes or even *name-SSN*. These links are refined by other matching transformations. The second kind of matching transformations calculates the similarity distance between every linked element. These transformations execute different matching heuristics (we explain them in the subsequent sections). In this case, the *name-name* link has a higher similarity value than *name-SSN* link. The third kind of matching transformation selects the links with best similarity values to produce a more accurate model with only a subset of links. For instance, we select only the *name-name* links. After the execution of these transformations, the weaving model can be manually modified in the weaving engine.

3.2.2 Transformation Generation

The transformation generation is the last phase in the production of model transformations. We implement higher-order transformations (HOT's) to interpret the different kinds of links captured by a weaving model. These HOT's generate the output model transformations. In other words, the weaving models are transformed into transformation models. The transformation model can be extracted into a textual language, for instance ATL or XSLT.

3.3 Matching Management

The matching management component (illustrated by the "bus" in Figure 3) controls the interactions between the transformation and the weaving engines. This component establishes the order in which the matching transformations are executed, and it synchronizes these transformations with the weaving engine. This way it is possible to manually update weaving models during the match process. The matching management component also provides facilities to inject models into a compatible format and to extract models into different formalisms.

4. WEAVING METAMODEL

The weaving metamodel specifies the different kinds of links that are generated by the matching transformations. Each kind of link corresponds to one transformation pattern. For instance, one of the most common patterns of declarative transformation rules is to match one class in a source model and to create a new class in a target model.

The weaving metamodels are created as extensions of a core weaving metamodel, as proposed in [7]. We illustrate an excerpt of this metamodel in Figure 4. The metamodel is written in KM3 [13]. KM3 is a simple textual language to define metamodels.

```
abstract class WLink extends WElement{
  reference child[*] container : WLink oppositeOf parent;
  reference parent : WLink oppositeOf child;
  reference end[1-*] container: WLinkEnd oppositeOf link;
}
abstract class WLinkEnd extends WElement{
  reference link : WLink oppositeOf end;
  reference element : WElementRef;
}
```

Figure 4. Excerpt of the core weaving metamodel

The *WLink* and *WLinkEnd* classes are the weaving elements that are extended more often, because these elements define the link types (*WLink*) and the linked elements (*WLinkEnd*). A *WLink* can

have child links to represent nested relationships, and it refers to one or multiple linked elements through the reference *end*.

4.1 Matching Metamodel Extensions

We show in Figure 5 an extension of this core weaving metamodel. The class *Element* is a concrete extension of *WLinkEnd*. It enables referring to any kind of (meta)model element. The class *Equivalent* contains two references to save the *source* and *target* elements. The class *Equivalent* has a similarity value that is calculated in the matching transformations. This value is a numeric value that measures the semantic proximity of the linked elements. The other classes capture five different transformation patterns:

- *Generic equality*: the class *Equal* indicates that the linked elements represent the same information.
- *Element binding*: the class *<Type>Binding* captures binding patterns between two model elements. The *<Type>* tag must be replaced by the element type, for example *AttributeBinding* or *ReferenceBinding*.
- *Attribute to references*: the class *AttributeToRef* captures links between attributes in the source model and references in the target model. The *targetAttribute* contains an attribute of the element referred by the *target* reference.
- *Element matching*: the class *ElementMatch* denotes the *from/to* link between a source and a target element.
- *Element inheritance*: the class *ElementInheritance* relates elements that inherit from others. The reference *super* points to the parent element of a given element.

```
class Element extends WLinkEnd { }
class Equivalent extends WLink {
  attribute similarity : Double;
  reference source container : Element;
  reference target container : Element;
}
class Equal extends Equivalent { }
class <Type>Binding extends Equivalent { }
class ElementMatch extends Equivalent { }
class AttributeToRef extends Equivalent {
  reference targetAttribute container : Element
}
class ElementInheritance extends Equivalent {
  reference super container : WLink;
}
```

Figure 5. Matching extensions

5. MATCHING TRANSFORMATIONS

In this section we present the different kinds of matching transformations in details. We define one generic model management operation for each matching transformation.

5.1 Creating Weaving Models

Transformations that create weaving models are the first kind of matching transformations that are executed. The model management operation that creates weaving models is called *CreateWeaving*. The operation takes two models M_a and M_b as input and transforms them into a weaving model M_w. M_a conforms to MM_a, M_b conforms to MM_b and M_w conforms to MM_w.

$$M_w : MM_w = CreateWeaving\ (M_a : MM_a, M_b : MM_b).$$

This operation matches a set of elements of a given type of M_a with a set of elements of a given type of M_b. It creates a restricted Cartesian-product $M_a \times M_b$. The operation creates a link between every pair of elements.

Figure 6 illustrates how the operation is implemented using a generic transformation rule. MM_a and MM_b denote the input metamodels. MM_w denotes the output weaving metamodel. This rule matches all elements of type *<TypeA>* with elements of type *<TypeB>* and produces an equivalence link between a source and target element.

```
rule CreateLink {
  from aSource : MMa!<TypeA>, aTarget : MMb!<TypeB>
  to alink : MMw!Equivalent (
    source <- aSource ,
    target <- aTarget
  )
}
```

Figure 6. Creation of equivalence links

The operation can also be modified to update weaving models (to create or to remove other links). In this case it has a weaving model as extra input parameter.

$$M_w : MM_w = CreateWeaving\ (M_a : MM_a, M_b : MM_b, M_w' : MM_w).$$

The use of matching transformations enables to change the types of the source or the target elements. This allows matching elements from different metamodels, for instance a KM3 *Class* with a SQL *Table*.

5.2 Calculating Element Similarity

The second kind of matching transformation calculates a similarity value between the elements referred by the *source* and *target* references, for every link of a weaving model. This similarity value is used to evaluate the semantic proximity between the linked elements. A link with a high similarity value indicates that there is a good probability that the source element must be translated into the target element.

We define a model management operation called *AssignSimilarity*. The operation takes a weaving model M_w' and a *weight* as input, and it produces a weaving model M_w as output. The input and the output models conform to the same weaving metamodel MM_w. The output weaving model has the new similarity values. However, there are many different methods to calculate similarities values. The tag *<method>* indicates the method that is implemented.

$$M_w : MM_w = AssignSimilarity\langle method\rangle\ (M_w' : MM_w, weight: int).$$

The *weight* parameter is used to restrict the similarity values between [0-weight]. This parameter enables to adjust the impact of a given similarity method. For instance, a similarity method that compares element names may have weight 0.8, and a similarity method that compares types may have weight 0.2. This means that the elements are considered more similar if they have the same name than the same type.

This operation executes update transformations, i.e., it does not create new links. Different matching transformations can be executed to obtain a more accurate similarity value. We implement element-to-element and structural methods. We explain them below.

5.2.1 Element-to-element Similarities

Element-to-element similarities are calculated taking the *source* and *target* elements of an *Equivalent* link and comparing the element names (or identifiers) in different ways. We implement different methods:

- *String similarity*: the names of the model elements are considered strings. The names are compared using string

comparison methods such as Levenshtein distance, n-grams and edit distance [5].

- *Dictionary of synonyms*: the names are compared using a dictionary of synonyms (we use *WordNet* [11]). This dictionary provides a tree of synonyms. The similarity between two terms (element names) is calculated according to the distance between these terms in the synonym tree. This way it is possible, for example, to increase the similarity value between elements such as *Teacher* and *Professor*, which does not yield good results if using string comparison methods.

However, some of these methods are already implemented and available in public APIs. We thus extend the ATL transformation engine to be able to call methods from external APIs. The transformation engine provides wrapper methods that can be applied to every model element, this way we are capable to use APIs such as the SimMetrics API [21] and the JWNL API [16].

5.2.2 Structural Similarity

Structural similarities are calculated using the internal properties of the model elements, e.g., types, cardinality, and the relationships between model elements, e.g., containment or inheritance trees. These data are encoded in the metamodels.

We implement a structural method called *metamodel-based similarity*. The metamodel-based similarity method is executed after an element-to-element method to improve the accuracy of these methods. The metamodel-based method calculates the similarity using the internal properties and the relationships between model elements.

5.2.2.1 Internal Properties

Model elements have a set of properties, such as type, cardinality, order, length, etc. Consider two model elements $a \in M_a$ and $b \in M_b$; M_a and M_b are different models, but conform to the same metamodel. A matching transformation compares the properties of *a* with the properties of *b*. If a given property has the same value, it adds 1(one) to a temporary similarity value. This temporary value is multiplied by the *weight* parameter and added to the initial similarity value. However, this generic comparison is valid only if M_a and M_b conform to the same metamodel. When the metamodels are different, the operation is adapted for every different property.

Consider two different metamodels, KM3 and SQL-DDL (the complete metamodels can be found at [1]). We consider two elements from these metamodels, *Attribute* from KM3 and *Column* from SQL-DDL. An *Attribute* has properties such as *type, lower, upper, isOrdered, or isUnique*. A *Column* has the following properties: *default, type, keys, canBeNull*. These properties cannot be directly compared if using a generic heuristic, because their values are not compatible and there is no name equivalence. For example, the transformation must take into account that *canBeNull* is a *Boolean*. The same information is captured analyzing the value of *lower* property.. We illustrate the transformation rule for this case in Figure 7.

This rule calculates the similarity between KM3 and SQL-DDL elements. It selects an *Equal* link that satisfies the following condition: the *source* reference points to an *Attribute* of a KM3 model, and the *target* reference points to a *Column* of the SQL-DDL model. The helper *requiredSim* compares the *required* property with the *CanBeNull* property, and returns one (1) if they satisfy the equality criteria.

```
rule UpdateStructuralSim {
   from mmw : MMw!Equal mmw.source.isTypeOf(KM3!Attribute)
          and mmw.target.isTypeOf(SQLDDL!Column))
   to alink : MMw!Equal (
       similarity <- ( mmw.similarity +
           mmw.source.requiredSim(mmw.target) ) * weight
     )
}
 helper context KM3!Attribute def: requiredSim
   (column : SQLDDL!Column) : Real =
   if (self.lower = 0 and column.canBeNull) or
     (self.lower = 0 and column.canBeNull)
     then 1
     else 0
   endif;
```

Figure 7. Structural similarity rule

5.2.2.2 Element Relationships

There are different kinds of relationships between elements of the same metamodel, for instance containment or inheritance relationships. Structural methods that exploit the element relationships rely on the following assumption: if two model elements are similar, the neighbors of these elements are likely to be similar as well. For example, if two attributes from two models have a high similarity value, the containing classes of these attributes have a good probability to be similar.

We create a heuristic inspired in the Similarity Flooding (SF) algorithm [18]. We implement a matching transformation that propagates the similarities values between related elements using the containment and the inheritance trees.

- *Containment tree*: it captures the containment relationships of a model. Consider for example a class with references and attributes. The nodes of the tree contain classes, attributes and references. They are all linked by containment edges.
- *Inheritance tree*: it captures the generalization relationships between the elements.

These methods can be executed in the same *AssignSimilarity* operation. However, it is also possible to have separate operations that are applied to specific models. For example, the inheritance tree is not relevant when creating a weaving model between SQL-DDL models that do not have native inheritance relationships.

These structures can be used to propagate the similarity between elements of different metamodels. Consider again the SQL-DDL and KM3 metamodels. The containment trees from both metamodels are different. However, the containment relationship between a *Table* and a *Column* is equivalent to the relationship between a *Class* and an *Attribute*. The matching transformations enable to build a containment tree of these two metamodels.

5.3 Selecting Best Links

The third kind of matching transformations selects only the links that satisfy a set of conditions. The selected links are included in the final weaving model. These matching transformations are generalized by the operation *Select<method>*.

$M_w : MM_w = Select<condition> (M_w' : MM_w)$.

The operation takes a weaving model M_w' as input and produce another weaving model M_w as output. Both weaving models conform to the same weaving metamodel MM_w. The *condition* tag denotes the selection criteria. Links are selected using two methods: link filtering and link rewriting. These methods are explained below.

5.3.1 Link Filtering

Link filtering methods selects only the links with the highest similarity values for every source element. This is because in model integration transformations it is necessary to translate all the elements of the source model (or as most as possible) into the target model. Thus, we want to obtain a link between every element of the source metamodel with the elements of the target metamodel. However, due to semantic differences, the target metamodel cannot always represent all the information from the sources.

We illustrate a matching transformation rule in Figure 8. This rule is executed for all the source elements. It loops over all the equivalence links (and inherited links) of a given source element and it selects the link that has the highest similarity value.

The *using* part declares variables to store an auxiliary similarity value and the selected link. The *allInstances()* method returns all the instances of links conforming to the *Equivalent* class. The *for* block selects the links that have the *source* reference equals to the *aSource* parameter. Then the similarity values are compared with a current similarity value. The maximum value and the corresponding link are stored in the auxiliary variables.

```
rule getMaxLink (aSource : MMa!ModelElement) {
  using {
    newLink : MMw!Equivalent = null;
    maxSim : Real = 0;
  } do {
      for(e in MMw!Equivalent.allInstaces()->
                select (e.source = aSource )) {
        if (e.similarity > maxSim) {
          maxSim <- e.similarity;
          newLink <- e;
        }
      }
    return newLink;
}
```

Figure 8. Link filtering method

The output weaving model contains one link for each element of the source model. This rule selects all the elements from the source model, but the same target element may be selected several times. The last adjustments are done by link rewriting methods.

5.3.2 Link Rewriting

Link rewriting methods analyze the relationships between links of a filtered weaving model. These relationships are used to transform simple links (e.g., *Equivalent*, *Equal*) into complex kind of links that capture different transformation patterns. Common patterns are nesting, inheritance, data conversions, concatenation, splitting, etc. For instance, if more than one source element is linked with the same target element through *Equal* links, this link can be rewritten as a *Concatenation* link. The most common form of link rewriting is the nesting between elements with containment relationships, for example classes and attributes, or tables and columns.

Consider a weaving model that links two KM3 metamodels, MM_a and MM_b. After the execution of a link filtering transformation, it contains a set of *ElementMatch* and *AttributeBinding* links. The class *ElementMatch* contains links between classes, and *AttributeBinding* contains links between attributes. However, they are children of the root element. Now consider classes $A \in MM_a$ and $B \in MM_b$, attributes $a \in A$, $b \in B$, links *ElementMatch (A, B)* and *AttributeBinding (a, b)*. Since *a* is an attribute of *A* and *b* is an attribute of *B*, the *AttributeBinding* link is rewritten as a link child of *ElementMatch*. Note that the rewriting is not based on the similarity values.

We illustrate the rewriting of nested links in Figure 9. This rule matches *AttributeBinding* and *ElementMatch* links at the same time and it checks if the *owner* of the attribute is the current element. If the result is true, it executes the rule and assigns the *class_link* element to the *attr_link.parent* reference.

```
rule NestedRewriting {
  from attr_link : MMw!AttributeBinding,
       class_link : MMw!ElemenentMatch (
      attr_link.source.owner = class_link.source and
      attr_link.target.owner = class_link.target )
  to link : MMw!AttributeBinding (
    parent <- class_link
  )
}
```

Figure 9. Rewriting of attribute-binding links

6. TRANSFORMATION GENERATION

The transformation generation is the last phase in the production of model transformations. This phase translates the weaving model produced by the matching transformations into a transformation model. We implement higher-order transformations (HOT's) to translate the extensions of *WLink*'s into transformation rules and bindings. A higher-order transformation is a transformation, such that the input and/or the output models are transformation models.

We describe below the input links (from Section 4.1) and the corresponding output transformation expressions (these expressions are described based on the ATL metamodel [15]). These HOT's are extensions of the pattern described at [8]. We illustrate them using the motivating example.

- *Equal*: these links are not translated into transformations. They are always rewritten by a link rewriting method.
- *ElementMatch*: the *source* reference is translated into an input pattern. An input pattern is the element after the *from* keyword. The *target* reference is translated into the first output pattern after the *to* keyword. For example, the transformation of class *Teacher* into class *Professor*.
- *<Type>Binding*: the *source* reference is translated into the source of a binding (i.e., the right expression after the "←" separator). The *target* reference is the target of a binding. For example, the binding between *SSN* attributes or *advisor* references.
- *AttributeToRef*: these links are translated into two bindings and one output pattern. The *target* reference is translated as the target expression of the first binding. The source of this binding is the new output pattern. This output pattern has the type of the *target* reference. This output pattern contains the second binding. The source of the second binding is the *source* reference, and the target is the *targetAttribute* reference. For example, the transformation of the *city*, *street* and *zip_code* into the *Address* class.
- *ElementInheritance*: the *source* and *target* references are translated in the same way of an *ElementMatch* link, i.e., a new transformation rule is created. If the element of the output pattern of another rule (i.e., generated from an *ElementMatch.target* reference) is referred by *super*, all the bindings of this referred rule are copied to the current rule. For example, the copy of *SSN* and *name* bindings into all the transformation rules of the motivating example.

7. DISCUSSION

The matching transformations are executed with two variations of the motivation example. In the first example, *MM1* and *MM2* conform to KM3. In the second example, *MM1* conforms to KM3 and *MM2* conforms to SQL-DDL. The weaving models are translated into model transformations. The goal is to verify if the transformations are generated correctly, and to verify if the matching transformations can be easily adapted in both examples.

The motivating example has different transformation patterns, such as class inheritance, nesting of elements, or classes with different names. *MM1* contains 17 elements and *MM2* contains 18. The creation of links between every model element without any type restriction yields a weaving model with 950 elements: 306 links, plus one right and one left element for each link, i.e., 3x306, plus additional control elements. It is important to reduce the number of initial links as early as possible in the process, to be able to scale up the approach to match larger models.

The weaving model with the type-restricted Cartesian product contains 273 elements, with 78 links. The name similarity method enables to match elements such as *SSN-SSN*, *name-name*, or *zip_code-code*. The dictionary of synonyms increases the similarity of elements such as *Professor* and *Teacher*, *Master* and *Student*. The containment tree enables to propagate the similarity of the attributes of *Master* and *Student*.

We execute the propagation of similarities two times. The propagation of the similarities more than two times increases the similarity between the classes (e.g., *Teacher* and *Professor*), but the values are not significantly different in our example. Several propagation steps may be more useful in the case of model comparison, where more accurate values are necessary.

The creation of links and the computation of similarities can be applied for more generic examples, not only to generate integration transformations. On the other side, the link filtering and rewriting methods are more specific to the type of the output.

Consequently, link selection methods are very important to obtain the final integration transformations. For example, the similarity between the abstract class *Person* and class *Professor* is high. This would produce a rule that transforms *Person* into *Professor*. The filtering method does not select links with abstract classes. Then, the link rewriting method copies the bindings of the attributes of the class *Person* to the rules that transform the inherited classes, i.e., *Master*, *Teacher*, *Undergraduate*.

The only link that is not generated correctly is *Undergraduate-Student*. This is because none of the initial similarity methods can find high similarity values. The values are not propagated, because the inheritance relationships exist only in the source model. We thus use the weaving engine to modify the weaving model. After applying all the transformations and using the weaving engine, the weaving model is reduced to 78 elements, with 12 links.

Finally, the weaving model is used as input to higher-order transformations. We created a HOT with 250 lines. It is relatively complex compared to the generated transformation, with only four ATL rules. However, this HOT and the matching heuristics are implemented to be used many times in different applications.

In the second example, we evaluate if the matching transformations can create weavings between models conforming to different metamodels. The base algorithms of the matching transformations are the same, leading to similar results. However, we adjust the implementation of the containment tree, as well as the metamodel-based heuristics. For example, we compare data types such as *String* (in KM3) and *char* (in SQL-DDL). Thus, the generic matching heuristics can be rapidly modified to match models conforming to different metamodels. The weaving models generated in both examples are equivalent.

To summarize, the use of matching transformations and weaving models enables to semi-automate the production of model transformations in an efficient manner. Matching transformations enables to rapidly implement different heuristics that produce a weaving model. These heuristics can be adapted to different metamodels. The weaving model captures different transformation patterns specified in a weaving metamodel. The weaving model is translated into a model transformation language.

8. RELATED WORK

To the best of our knowledge, our solution is the only complete model driven approach that automates the production of model transformations. However, there has been extensive work on how to find relationships between schemas and ontologies. These approaches have different goals, such as data and schema integration [2] [6] [17], ontology merging and alignment [10] [19] [20], or ontology integration [9]. Amongst these several approaches, COMA++ [2] and the API of Euzenat [10] are the solutions most similar to ours.

COMA++ implements a set of heuristics, using for instance element-to-element methods or incremental matching. COMA++ provides an interactive user interface to combine these heuristics. Matching transformations provide a more suitable mechanism for adaptation, because the declarative nature of the transformations allows abstracting implementation details, such as the creation of new elements and the match of several elements.

The work of Euzenat factors out schema matching features and proposes a generic API. The API provides interface methods used to implement different matching heuristics and to combine different matching results. The main drawback of this API is that the new matching methods must be implemented almost from scratch. The API does not provide interfaces for each different matching phase. We implement these operations using model transformations. This enables to create customized heuristics to match different metamodels. However, we do not provide operations to evaluate different matching results.

The solution from [9] proposes machine learning techniques to select amongst a set of heuristics, and not to create heuristics as in most part of solutions. We believe this is a complementary approach. The machine learning techniques could be enhanced to support our matching transformations as input.

iMAP [6] is one of the few approaches that creates complex links. However, the links are all created in the beginning of a matching (equivalent to our *CreateWeaving* operation). We create complex mappings after filtering a weaving model. Our link rewriting method creates a smaller number of complex links that are targeted to produce model transformations.

Similarity Flooding (SF) [18] is a generic structural heuristic that propagates the similarity of a pair of nodes through their neighbors. However, as the author say in [18], it is not adapted to

match models conforming to different metamodels. This algorithm is the basis of our metamodel-based matching transformations. We improve this heuristic to support the matching of different metamodels, with two different ways to propagate the similarities through the neighbors' elements.

9. CONCLUSIONS

In this paper, we have presented a solution to automate the production of model transformations. We have proposed to use matching transformations that create weaving models. These transformations execute different matching heuristics. The weaving models capture common transformation patterns between model elements. The weaving model is translated into executable model integration transformations.

We have shown that matching transformations are a practical solution to implement new or to adapt matching heuristics. The use of declarative transformation languages enabled to abstract implementation details on how to apply these heuristics. The separation of the whole matching process into different kinds of matching transformations allowed combining different methods in a straightforward way.

The matching transformations enabled the creation of weaving models between models conforming to different metamodels, and also the creation of links only between a restricted set of elements. We proposed a new metamodel-based matching transformation that takes advantage of every property of a given metamodel. We have presented a new link rewriting operation that analyzes the relationships between the links of a weaving model. These links are transformed into complex kind of links. This operation is particularly important to capture complex transformation patterns.

We developed a weaving metamodel that captures common transformation patterns. The weaving models conforming to this metamodel were translated into transformations using higher-order transformations. We were able to produce the transformation model that performs the data translation from a source into a target model.

The use of several matching transformations can cause performance problems when generating transformations between large models. Thus, the optimization of these operations is becoming important and is a subject for future work. For instance, after choosing a set of operations to create a weaving model, these operations could be merged by a transformation engine to be executed in a single rule.

10. ACKNOWLEDGMENTS

This work is partially supported by ModelPlex project. We would like to thank the members of the ATLAS team.

11. REFERENCES

[1] AM3 Atlantic Zoo. Reference site: http://www.eclipse.org/gmt/am3/zoos/atlanticZoo/. Oct. 2006

[2] Aumueller, D, Do, H H, Massmann, S, Rahm, E. Schema and ontology matching with COMA++. In proc. of SIGMOD 2005. pp 906-908

[3] Balogh, A, Németh, A, Schmidt, A, Ráth, I, Vágó, D, Varró, D, Pataricza, A. The VIATRA2 model transformation framework. In proc. of ECMDA 2005 - Tools Track, 2005

[4] Bernstein, P A. Applying Model Management to Classical Meta Data Problems. In proc. of CIDR 2003, pp 209-220

[5] Cohen, W, Ravikumar, P, Fienberg, S E. A Comparison of String Distance Metrics for Name-Matching Tasks. In proc. of IIWeb 2003, pp 73-78

[6] Dhamanka, R, Lee Y, Doan, A, Halevy, A, Domingos P. iMAP: Discovering Complex Semantic Matches between Database Schemas. In proc. of SIGMOD 2004, pp 383-394

[7] Didonet Del Fabro, M, Bézivin, J, Jouault, F, Valduriez, P. Applying Generic Model Management to Data Mapping. In proc. of BDA 2005, Saint-Malo, France, pp 343-355

[8] Didonet Del Fabro, M, Bézivin, J, Valduriez, P. Model-Driven Tool Interoperability: An Application in Bug Tracking. In proc. of ODBASE'06, LNCS 4275, Nov. 2006, pp 863-881

[9] Ehrig, M, Staab, S, Sure, Y. Bootstrapping Ontology Alignment Methods with APFEL. In proc. of the 4th ISWC 2005, Galway, Ireland, volume 3729 of LNCS, pp 186-200

[10] Euzenat, J. An API for Ontology Alignment. In proc. of ISWC 2004, pp 698-712

[11] Fellbaum, C. WordNet, an Electronic Lexical Database. MIT Press, 1998. Reference site: http://wordnet.princeton.edu/

[12] Gardner, T, Griffin, C, Koehler, J, Hauser, R. A review of OMG MOF 2.0 QVT submissions and recommendations towards the final standard. 1st International Workshop on Metamodeling for MDA, York, UK, 2003

[13] Jouault, F, Bézivin, J. KM3: a DSL for Metamodel Specification. In proc. of 8th FMOODS, LNCS 4037, Bologna, Italy, 2006, pp 171-185

[14] Jouault, F, Kurtev, I. On the Architectural Alignment of ATL and QVT. In proc. of the 2006 ACM Symposium on Applied Computing (SAC 06). ACM Press, Dijon, France, 2006, chapter Model transformation, pp 1188-1195

[15] Jouault, F, Kurtev, I. Transforming Models with ATL. In proc. of the Model Transformations in Practice Workshop at MoDELS 2005, Montego Bay, Jamaica, pp 128-138

[16] JWNL (Java WordNet Library). Reference site: http://sourceforge.net/projects/jwordnet. August 2006

[17] Madhavan, J, Bernstein, P A, Rahm, E. Generic Schema Matching Using Cupid, In proc. of VLDB 2001, pp 49-58

[18] Melnik, S. Generic Model Management: Concepts and Algorithms, Ph.D. Dissertation, University of Leipzig, Springer LNCS 2967, 2004

[19] Mitra, P, Wiederhold, G, Kersten, M. A graph-oriented model for articulation of ontology interdependencies. LNCS, 1777:86+, 2000

[20] Noy, N, Musen, M. PROMPT: Algorithm and Tool for Automated Ontology Merging and Alignment. In proc. of. AAAI/IAAI, pp 450–455

[21] SimMetrics. Developed by Sam Chapman. Reference site: http://sourceforge.net/ projects/simmetrics/. August 2006

[22] Shvaiko, P, Euzenat, J. A Survey of Schema-Based Matching Approaches. JoDS IV, pp 146-171, 2005