Softw Syst Model
DOI 10.1007/s10270-008-0094-z

SPECIAL SECTION PAPER

Towards the efficient development of model transformations using model weaving and matching transformations

Marcos Didonet Del Fabro · Patrick Valduriez

Received: 2 July 2007 / Revised: 29 April 2008 / Accepted: 13 May 2008

Abstract Model transformations can be used in many different application scenarios, for instance, to provide interoperability between models of different size and complexity. As a consequence, they are becoming more and more complex. However, model transformations are typically developed manually. Several code patterns are implemented repetitively, thus 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 present a novel approach that uses matching transformations and weaving models to semi-automate the development of transformations. Weaving models are models that contain different kinds of relationships between model elements. These relationships capture different transformation patterns. Matching transformations are a special kind of transformations that implement methods that create weaving models. We present a practical solution that enables the creation and the customization of different creation methods in an efficient way. We combine different methods, and present a metamodel-based method that exploits metamodel data to automatically produce weaving models. The weaving models are derived into model integration transformations. To validate our approach, we present an experiment using metamodels with distinct size and complexity, which show the feasibility and scalability of our solution.

Communicated by Dr. Jean Bezivin.

M. Didonet Del Fabro (✉)
ILOG SA, Paris, France
e-mail: mddfabro@ilog.fr

M. Didonet Del Fabro · P. Valduriez
ATLAS Group, INRIA& LINA, Nantes, France
e-mail: patrick.valduriez@inria.fr

Keywords Model engineering · Matching transformations · Model weaving

1 Introduction

Model transformations are a central component in model driven engineering practices. Many model transformation languages have emerged from industrial and academic research [4,18,20,21]. 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 of these expressions consist of 1:1 relationships or other common patterns, such as nesting or concatenation.

These transformations are often created manually. To the best of our knowledge, there is no Model Driven Engineering (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, diminishes the errors that may occur in manual coding, and increases the quality of transformational code.

The discovery of transformation patterns to integrate models is closely related to schema and ontology matching

approaches (see the survey in [33]). These approaches aim at discovering semantic relationships between elements of different schemas or ontologies [8,13,29]. However, these approaches have some drawbacks. Most 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 (methods) is too big. This makes it difficult to decompose and to customize different methods. 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 present the execution of matching transformations, transformations that select a set of elements from a set of input metamodels and produce links between these elements. As we presented in [10], these links are saved in a weaving model. A weaving model conforms to the extensions of a weaving metamodel. A weaving model contains abstract and declarative links that are used to produce model integration transformations. Model integration transformations are used in standard interoperability applications.

The overall process is the following:

- the designer provides the input metamodels MM_A and MM_B;
- a sequence of matching transformations is executed. They produce a weaving model with a set of links between MM_A and MM_B;
- the designer verifies the links (and correct them, if necessary);
- a transformation model is generated based on the set of links;
- the transformation produced is used to transform a model conforming to MM_A into a model conforming to MM_B.

Matching transformations enable to rapidly implement or adapt methods to create weaving models. We propose a metamodel-based method that exploits the internal features of the set of input metamodels to produce weaving models. This method 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 methodology to semi-automate the development of model transformations. The methodology is based on MDE best practices, which enables the rapid development and reuse of different methods. The main innovation is to use matching transformations to allow an easy development of different matching methods. We propose a metamodel-based method that exploits the information from the set of input metamodels and from the weaving metamodel. These matching transformations automatically create weaving models. The matching transformations can be combined and parameterized using configuration models. The configuration models can be shared among developers to reuse the execution parameters. To validate our approach, we present an experiment using large metamodels, showing the feasibility and scalability of our approach in real world scenarios.

This paper extends our previous work [11] with three major improvements. First, we define more precisely our metamodel-based method and we present a method for storing model elements that are not matched using this method. Second, we present a configuration model that enables to easily configure the execution of chains of matching transformations. Finally, we present an experiment using large metamodels, in a metamodel comparison application scenario. These experiments demonstrate the feasibility and scalability of our approach.

This paper is organized as follows. Section 2 gives the motivating example. Section 3 presents the core MDE concepts used in this paper. Section 4 presents the general process of the production of model transformations. Section 5 presents weaving metamodel extensions that capture different kinds of relationships between models. Section 6 describes the matching transformations in more details. Section 7 describes the experiments. Section 8 presents the related work. Section 9 concludes.

2 Motivating example

We motivate the need to automatically create model transformations using two simple metamodels *MM1* and *MM2*. Both metamodels are illustrated in Fig. 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 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 that the student is undergraduate or master. The address of the professors and the students is factored out on the class *Address*.

In Fig. 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

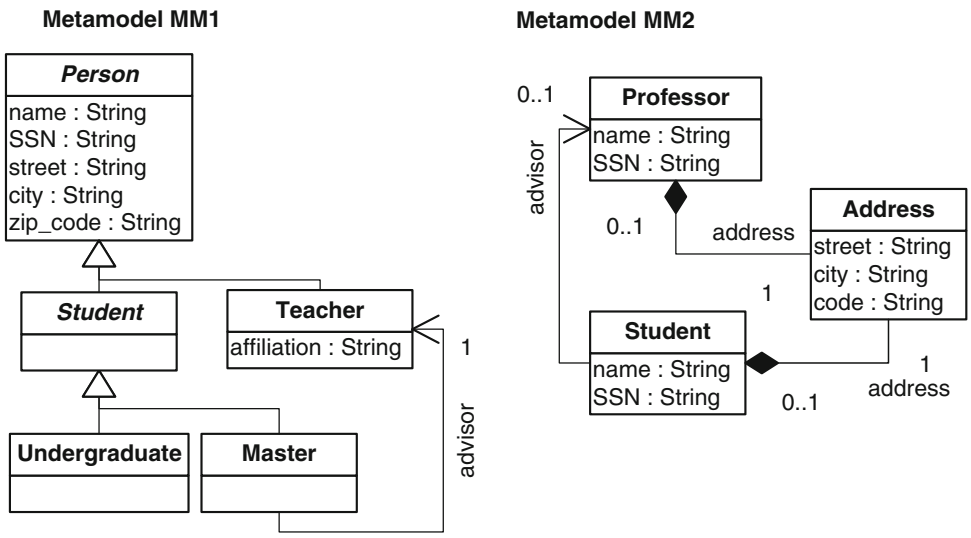

Fig. 1 Two simple metamodels

```
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
  )
}
```

Fig. 2 Transformation definition

language is available in [21]). We choose ATL because it provides a simple syntax adapted for model transformations.

This transformation has three rules; while executing the transformation, each rule matches one element of the source model and creates elements in the target model. In a typical development scenario, the developer creates the transformation manually. He must know that *Teacher* is transformed into *Professor* and that *Master* and *Undergraduate* are transformed into *Student*. Then, all the attributes and references of each class must be translated as well (*name*, *SSN*, *address*, *advisor*, *street*, *code*, etc.).

This transformation has 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 comparison scenarios. Despite this example being simple, it is easy to envisage the creation of transformations between very large source and target metamodels. The same kinds of expressions presented would be coded several times. This is a known problem when developing large model integration applications. Many of them are not trivial, for instance, to handle elements *Undergraduate* and *Master*. 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 elements. These links must

be saved in another model. This model can be validated or modified by the transformation developer (see Appendix A for a complete example).

Methods already used in ontology and schema matching solutions can be applied 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 a simple methodology that enables the implementation of new methods and the adaptation of existing ones in an efficient way. 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 Model driven engineering

In this section we present the central MDE definitions used in this paper. The basic assumption in MDE is to consider models as first-class entities (following [9]).

Definition 3.1 (Directed multigraph) A directed labeled multi-graph $G = (N_G, E_G, \Gamma_G)$ consists of a finite set of nodes N_G and a finite set of edges E_G, a mapping function $\Gamma_G : E_G \rightarrow N_G \times N_G$.

Definition 3.2 (Model) A model $M = (G, \omega, \mu)$ is a triple where:

- $G = (N_G, E_G, \Gamma_G)$ is a directed multigraph,
- ω is itself a model associated to a multigraph $G_\omega = (N_\omega, E_\omega, \Gamma_\omega)$,
- $\mu : N_G \cup E_G \rightarrow N_\omega$ is a function associating elements (nodes and edges) of G to nodes of G_ω. The function μ associates every node and edge of G ($N_G \cup E_G$) with one element in $\omega(N_\omega)$.

Definition 3.3 (Reference model) Given a model $M_1 = (G_1, \omega_1, \mu_1)$, and a model $M_2 = (G_2, \omega_2, \mu_2)$, if $\omega_1 = M_2$, M_2 is called the reference model of M_1.

Some models are their own reference model ($\omega = M$). This allows stopping the recursion introduced in this definition. The relation between a model and its reference model is called *conformance*. However, we observe from different domains (XML, RDBMS, ontologies) that only three levels are needed. We call these three levels metametamodel (M3), metamodel (M2) and terminal model (M1). A *metametamodel* is a model that is its own reference model. A *metamodel* is a model such that its reference model is a metametamodel. A *terminal model* is a model such that its reference model is a metamodel.

In order to capture the relationships (i.e., links) between model elements, we propose using weaving models. A weaving model conforms to a weaving metamodel. The weaving metamodel defines the kinds of links that may be created.

Definition 3.4 (Weaving metamodel) A weaving metamodel is a model $MM_W = (G_M, \omega_M, \mu_M)$, that defines link types, such that:

- $G_M = (N_M, E_M, \Gamma_M)$,
- $N_M = N_L \cup N_{LE} \cup N_O$, N_L denotes the *link types*; N_{LE} denotes the *link endpoint types* and N_O denote other auxiliary nodes,
- $\Gamma_M : E_M \rightarrow (N_L \times N_{LE}) \cup (N_O \times N_M)$, i.e., a *link type* refers to multiple *link endpoint types* and the auxiliary nodes refer to any kind of node.

Definition 3.5 (Weaving model) A weaving model is a model $M_W = (G_W, \omega_W, \mu_W)$, a graph $G_W = (N_W, E_W, \Gamma_W)$, such that its reference model is a weaving metamodel ($\omega_W = MM_W$).

We present a core weaving metamodel based on the previous definitions. The metamodel is illustrated in Fig. 3. The core metamodel has elements with information about link types, link endpoints and element identifications. Element identification is a practical solution for saving unique identifiers for the linked elements. These values are used by a function to access the elements of the linked model elements.

- *WElement* is the base element from which all other elements inherit.
- *WModel* represents the root element that contains all model elements. It is composed by the weaving elements and the references to woven models.
- *WLink* expresses a link between model elements, i.e., it has a simple linking semantics.
- *WLinkEnd* defines the link endpoint types. Every link endpoint represents a linked model element. It allows creating N-ary links.
- *WElementRef* elements are associated with a dereferencing function. This function takes as parameter the value of the *ref* attribute and it returns the linked element. For practical reasons, we define a string attribute.

Weaving models enable the creation of abstract links between model elements, though they are not executable. Executable operations between models are implemented using model transformations.

Definition 3.6 (Model transformation) A model transformation is an operation that given as input a set of models, evaluates their elements and produces as output a set of models.

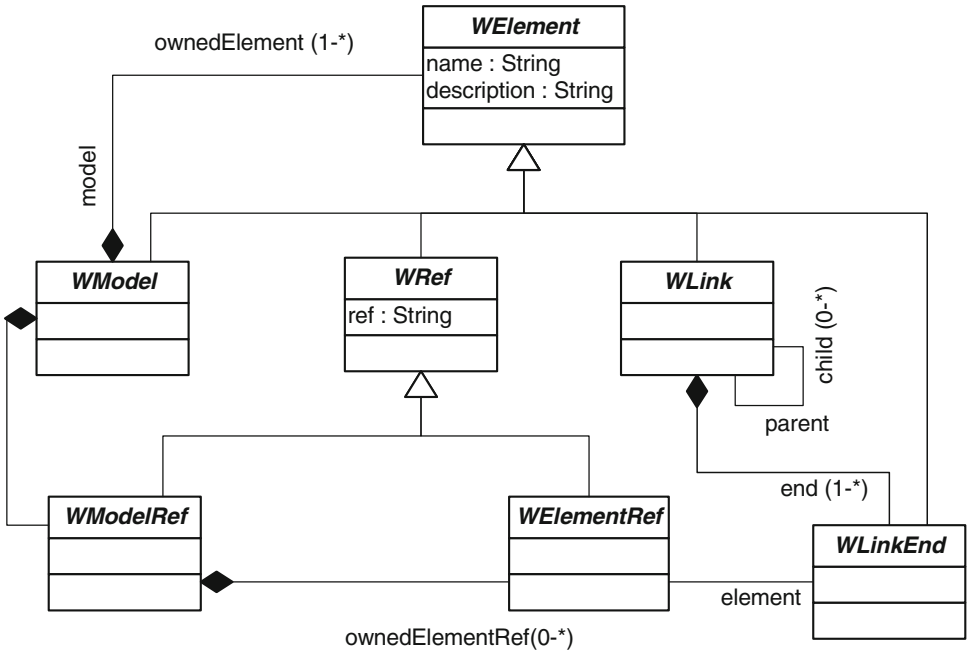

Fig. 3 Core weaving metamodel

A model transformation has the following signature:

$$\langle \mathrm{OUT}_1 : \mathrm{MM}_{\mathrm{OUT1}}, \ldots, \mathrm{OUT}_\mathrm{m} : \mathrm{MM}_{\mathrm{OUTm}} \rangle = \mathrm{T}(\langle \mathrm{IN}_1 : \mathrm{MM}_{\mathrm{IN1}}, \ldots, \mathrm{IN}_\mathrm{n} : \mathrm{MM}_{\mathrm{INn}} \rangle)$$

T is the operation name; $\langle \mathrm{IN}_1 - \mathrm{IN}_n \rangle$ is the set of input models ($n \geq 1$); the input models conform to the input metamodels $\langle \mathrm{MM}_{\mathrm{IN1}} - \mathrm{MM}_{\mathrm{INn}} \rangle$; the input metamodels may be equal; $\mathrm{OUT}_1 - \mathrm{OUT}_m$ is the set of output models ($M \geq 1$); the output models conform to the output metamodels $\langle \mathrm{MM}_{\mathrm{OUT1}} - \mathrm{MM}_{\mathrm{OUTn}} \rangle$; the output metamodels may be equal.

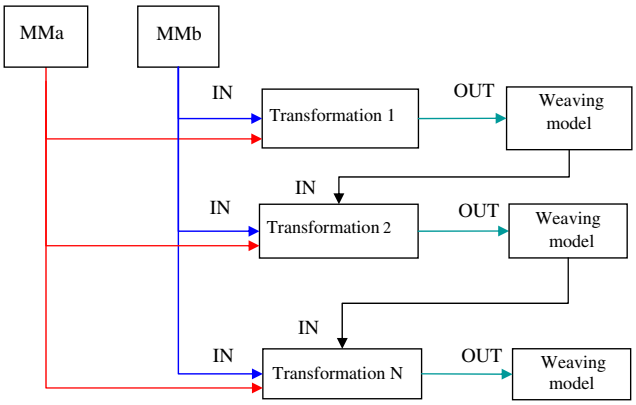

Fig. 4 Creation and refinement of weaving models

4 General overview

In this section, we present a general overview of our approach to automate the production of model transformations. There are three major phases: weaving metamodel specification, creation of weaving models and transformation production.

4.1 Weaving metamodel specification

To capture various transformation patterns, we define different kinds of links in a weaving metamodel. For instance, the motivation example contains different transformation patterns, such as class inheritance, nesting of elements, or classes with different names. The weaving metamodels are created as extensions of a core weaving metamodel. Each kind of link corresponds to one transformation pattern. For instance, one of the most common patterns of declarative transformation rules is to create a new *Class* in the source model for every *Class* in the target model. The creation of the weaving metamodel is a manual task, based on design decisions. Consequently, its specification is a key process of our approach.

4.2 Creation of weaving models

The second phase is the creation of weaving models with the concrete links between the input metamodels. We have two metamodels as input, which are used to produce one weaving model as output. This weaving model has different kinds of links. we propose using model transformations to produce the weaving model.

However, there is not a unique solution/transformation that produces a weaving model between the two input metamodels, with all the desired links. The process is iterative. Several methods are executed sequentially, as shown in Fig. 4.

Each one takes as input a weaving model and the two metamodels as input, and then produces another weaving model as output, which is in turn taken as input by another method. The links of the weaving model are refined after the execution of each transformation.

However, it is not always possible to create a weaving model with only correct links between the model elements. For instance, a transformation that performs a Cartesian product between two metamodels creates links between *name-name* attributes and *name-SSN* attributes. A second transformation may compute a similarity distance between every linked element (a value between 0 and 1), to select only links with similar name (we explain different methods in the subsequent sections). Different similarity values enables setting the impact of the transformations in the overall process, i.e., the higher similarity values have a higher impact. In this case, the *name-name* link has a higher similarity value than the *code-zip_code* link.

Then, we select the links with best similarity values to produce a more accurate weaving model with only a subset of links. After the execution of these transformations, the final weaving model can still be manually modified, because these methods cannot discover all the possible links. 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, and thus a link between these elements is created. This link is used to produce a rule that transforms *Person* into *Professor*.

4.3 Producing transformations

The production of transformations is the last phase in the overall process. We implement higher-order transformations (HOT's) that interpret the different kinds of links captured by a weaving model, as shown in Fig. 5. These HOT's take as input the weaving model, and produce a model transformation. The weaving model conforms to a weaving metamodel MM_w and the transformation models conform to a transformation metamodel (the conformance relation is indicated by $c2$). In other words, the weaving models are transformed into transformation models. The transformation models can be extracted into a textual language, for instance ATL or XSLT, to be executed in a specific transformation engine. The produced transformation is used to transform the terminal models conforming to the input metamodels (e.g., MM_1) into the terminal models conforming to the output metamodels (e.g., MM_2).

The whole process of producing model transformations is completely based on MDE concepts. In the following sections, we explain in detail these different phases.

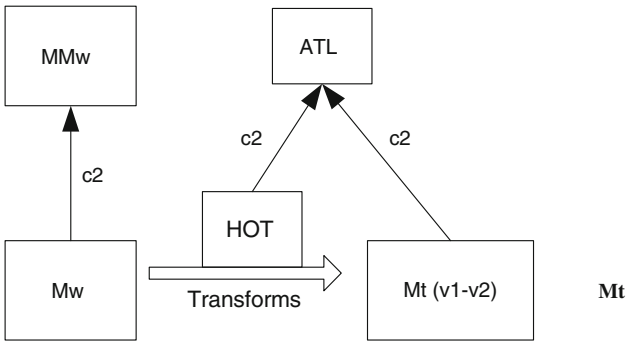

Fig. 5 Producing a model transformation

5 Metamodel extensions for matching

In this section, we present an extension to the core weaving metamodel (see Fig. 6) that captures a set of common transformation patterns. 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 computed by the transformations. This value is a numeric value that measures the semantic proximity of the linked elements. The other classes capture five different transformation patterns. In order to support additional patterns, for instance, multiple inheritances or the concatenation/union of elements, it is necessary to create different metamodel extensions, following the same approach.

- **Generic equality**: the class *Equal* indicates that the linked elements represent the same information.
- **Element binding**: the class $\langle Type \rangle Binding$ captures binding patterns between two metamodel elements. The $\langle Type \rangle$ tag must be replaced by the type of the element that is linked. For example, *AttributeBinding* when linking attributes, or *ReferenceBinding* for references.
- **Attribute to references**: the class *AttributeToRef* captures links between attributes in the source model and

```
class Element extends WLinkEnd {}
class Equivalent extends WLink {
    attribute similarity : Double;
    reference source container : Element;
    reference target container : Element;
}
class <Type>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;
}
```

Fig. 6 Metamodel extensions

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.

6 Matching transformations

In this section, we present the different kinds transformations that produce weaving models. We define one generic model management operation for each kind of transformation.

Definition 6.1 (Matching) Matching is the process of establishing relationships between elements belonging to different models.

The whole process is encapsulated in an operation called Match.

Definition 6.2 (Match) 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.

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

This definition allows creating elements between metamodels or terminal models. The matching process uses different techniques to create links between a set of model elements. The goal is to discover the relationships between a set of input models and to create a weaving model. We implement the match operation using model transformations. This means that the matching techniques are implemented as domain-specific model transformations. These domain-specific transformations are called matching transformations.

Definition 6.3 (Matching transformation) A matching transformation is a domain-specific transformation T that takes two or more models as input, and that transform them into a new weaving model M_W.

$$\langle M_{W1} : MM_{W1}, \ldots, M_{Wn} : MM_{Wn} \rangle = T(\langle IN_1 : MM_{IN1}, \ldots, IN_m : MM_{INm} \rangle)$$

A matching transformation implements different methods that produce weaving models. We may consider that the set of input models are transformed in a weaving model. A weaving can also be the input of a matching transformation. We first execute a matching transformation that creates an initial weaving model. This weaving model is refined (in order to improve its accuracy) using additional matching transformations. We explain the different kinds of matching transformations in the following sections.

6.1 Creating weaving models

Transformations that create weaving models are the first kind of matching transformations. The model management operation that creates weaving models is called *CreateWeaving*. The transformation takes two models *Ma* and *Mb* 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 with the same type.

Figure 7 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 $\langle TypeA \rangle$ with elements of type $\langle TypeB \rangle$ and produces an equivalence link between a source and target element.

The operation can also be modified to update weaving models (to create or 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 weaving model with the type-restricted Cartesian Product produces more links than necessary. The name similarity method enables to match elements such as *SSN-SSN,* or *name-SSN*. These links are further refined using different transformations.

6.2 Calculating element similarity

These matching transformations compute 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.

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

Fig. 7 Creation of equivalence links

To compute the similarities, 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 compute similarities values. The tag $\langle method \rangle$ 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]. If several transformations are executed, the weights are normalized into 1, enabling a better comparison of results between different methods. This parameter enables adjusting 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 as the same type. 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.

6.2.1 Element-to-element similarities

Element-to-element similarities are computed 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 [7].
- **Dictionary of synonyms:** the names are compared using a dictionary of synonyms (we use WordNet [16]). This dictionary provides a tree of synonyms. The similarity between two terms (names of the model elements) is computed 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 when 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, such as the SimMetrics API [32] and the JWNL API [22].

6.2.2 Structural similarity

Structural similarities are computed 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.

6.2.2.1 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, a temporary similarity value is increased by 1. 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 [19] and SQL-DDL (the complete metamodels can be found in [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 method, 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 by analyzing the value of *lower* property. We illustrate the transformation rule for this case in Fig. 8.

This rule computes the similarity between KM3 and SQL-DDL elements. It is a domain-specific rule, i.e., it is manually designed and it is used uniquely with KM3 and SQL-DDL metamodels. 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 a SQL-DDL model. The helper *requiredSim* compares the required property with the *CanBeNull* property, and returns 1 if they satisfy the equality criteria.

6.2.2.2 Element relationships There are different kinds of relationships between elements of the same metamodel, for instance, containment or inheritance relationships. Most existing structural methods that exploit the element relationships

```
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) then
      1
      else
      0
      endif;
```

Fig. 8 Structural similarity rule

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 a link between two attributes of two different metamodels has a high similarity value, the containing classes of these attributes have a good probability to be similar.

We create a method inspired by the Similarity Flooding (SF) algorithm [25]. We briefly explain the key idea of SF, and then how we improve it. Consider two input metamodels $M_a$and M_b, and the model elements $a, a' \in M_a$ and $b, b' \in M_b$. Elements a and a' are connected by an edge *(a, "containment", a')*. Elements b and b' are connected by an edge *(b, "containment", b')*. Initially, the algorithm executes a Cartesian product $M_a \times M_b$ and assigns a similarity value for every pair of elements. Consider the pairs (a, b) and *(a', b')*, with similarities x and y, respectively. We can see that a is related to a' by a "*containment*" edge. The same is valid for b and b'. The key idea of SF is to propagate the similarity value between the pair of elements that are connected by edges with the same label. In other words, it propagates x to (b, b') and updates the similarity value y.

Now, we explain our improvement to this solution. The main advantage is the possibility of having different forms of propagations based on different structural or semantic relationships between the elements of the input metamodels, and not based only on the value of the label of the edges. Assuming that the similarities are propagated between elements connected by the same label is too restrictive, because it cannot capture different semantic relationships between models. In contrast, it is also too generic, because we cannot create application-specific propagation models.

The propagation graph is encoded in a weaving model, called weaving propagation model. The description of the weaving propagation model is one of the extensions presented in this paper. The weaving model conforms to the metamodel extension shown in Fig. 9 (it is written using KM3).

The class *WAssociation* is an abstract class that depicts relationships between extensions to *WLink*s within the same weaving model. The class *PropagationElement* has two references: *outgoingLink* refers to the link with the source similarity value, and *incomingLink* refers to the link with the target similarity value. The *propagation* attribute contains a value that is multiplied with the similarity value of the *outgoing Link*.

Our approach allows constructing different weaving propagation models and also propagating similarities between elements conforming to different metamodels. The central issue is how to create relevant propagation elements and propagation values between a set of links. We show how this is done using the generic transformation rule from Fig. 10. This rule assumes that the input model of the transformation is a weaving model (*AMW*) that contains a set of links with a similarity measure.

The rule input pattern matches two links. These links are extensions of *Equivalent* links. The source link contains the similarity value that is propagated. The target link contains the similarity that is updated. This means the similarity is propagated from the source link to the target link. The $\langle semanticguard \rangle$ determines the condition that must be filled to create a propagation element (we show different semantic guards later). These semantic guards are specific for every different relation between the elements. The rule creates a propagation element and assigns the source and

```
package mw_core {
        class WAssociation extends WElement {
        }
}
package mmw_propagation {
        class PropagationElement extends WAssociation {
                reference incomingLink : Equivalent;
                reference outgoingLink : Equivalent;
                attribute propagation : Double;
        }
}
```

Fig. 9 Weaving propagation metamodel extension

```
rule CreatePropagationElement {
  from
   source_link : AMW!Equivalent,
         target_link : AMW!Equivalent (
          <semantic guard>
         )
    to
 out : AMW!PropagationElement (
    propagation <- 1 / <propagation_value>,
    outgoingLink <- source_link,
    incomingLink <- target_link
}
```

Fig. 10 Creation of propagation edges

target links to the corresponding references. The propagation value is computed in this rule.

We develop three different kinds of propagation based on this generic rule. We illustrate our approach assuming that the input metamodels conform to KM3. However, the rules can be adapted to match different metamodels or models.

Containment-tree propagation: the containment tree propagation method enables propagating the similarity between elements that have containment relationships, for instance, classes and attributes or classes and references (note that this is not the containment between classes, but between classes and its members). Consider for example a KM3 *Class*. The reference *structuralFeatures* points to classes *Reference* and/or *Attribute*. We create propagation elements from the links between classes to the links between its attributes. The guard of the rule is shown below. The *getReferredLeft/Right* is a helper that returns the element of the input metamodel.

```
source_link : AMW!ElementEqual,
target_link : AMW!AttributeEqual (
  attr_link.getReferredLeft.owner = class_link.getReferredLeft
  and
  attr_link.getReferredRight.owner = class_link.getReferredRight
)
```

The link between classes is assigned to the *outgoingLink*, and the link between the attributes is assigned to the *incomingLink*. In the same way as the SF algorithm, we consider that a given method can contribute to a maximum similarity value of 1. Consequently, the propagation value is 1 divided by the multiplication of the total number of attributes of the two input classes.

Relationship-tree propagation: this propagation method takes into account the type of the references of two given classes. For instance, consider the links between classes (a, b) and (c, d); a has a reference to c and b has a reference to d. The relationship graph is used to propagate the similarity between these two links.

Inheritance-tree propagation: this method enables propagating the similarity value from the link between two source classes to links between the parent classes of the source classes, if any. It can be considered as an extension to the relationship tree propagation method. However, it takes into account only the references that represent inheritance relationships. In KM3, this reference is called *supertypes.*

These propagation elements are created in the same weaving model. However, it is also possible to have separate weaving models that are used with specific input models. For example, the inheritance tree propagation is not relevant when creating a weaving model between SQL-DDL models that do not have native inheritance relationships. Thus, this propagation method is not used in this particular matching scenario.

These structures can be used to propagate the similarity between elements of different metamodels as well. 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.

6.3 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 $Select\langle method\rangle$.

$$M_w : MM_w = Select\langle condition\rangle(M_w{}' : MM_w).$$

The operation takes a weaving model M'_w as input and produces 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.

6.3.1 Link filtering

There are different kinds of link filtering methods. The simplest method (and also most used) is to set up a minimum threshold value and to select only the links that have a similarity value higher than this threshold. The biggest drawback of this method is the choice of a correct threshold method. Creating a new weaving model based on low threshold values may yield too much false links, i.e., that should not be created. In contrast, too high threshold values may filter relevant links.

We implement a link filtering method that selects only the links with the highest similarity values for every source element. This is because, in a model integration transformation, it is necessary to translate all the elements of the source model (or as most as possible) into the target model. However, due to semantic differences, the target metamodel cannot always represent all the information from the sources. If one element has more than one link with the same similarity values, the filtering methods select all links, enabling to have more than just one to one relations. This is a simplifying assumption, but different methods could be implemented in such cases.

6.3.2 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 links that capture different transformation patterns.

Common patterns are nesting, inheritance, data conversion, concatenation, splitting, etc. For instance, if more than one source element is linked with the same target element through an *Equal* link, 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. 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.

These transformations are executed always after the computation of some similarity estimation. The different guards in the transformation rules match the existing links. The *to* part enables to recreate the same links (to copy them) or to create new kinds of complex links.

Link rewriting transformations are closely related to the application scenario. These methods use more specific kinds of links (e.g., not only equality or equivalence), which means they are less generic than similarity calculation method.

In addition to the creation of complex links, link rewriting transformations can create links that record different kinds of information about the overall matching process. After the execution of a set of matching transformations, it is normal that some elements of the source metamodel are not linked with any element of the target metamodel, and vice-versa. We create a link rewriting transformation that records the source and/or target elements that are not referenced by any link. This kind of link can be used for different purposes: to verify if the resulting weaving model is correct, to record which elements cannot be translated from one model to another, or to use them as input to algorithms that calculate the difference between models. Figure 11 depicts an extension to the core weaving metamodel that record elements that are not linked.

The class *NotFound* (extension to *WLink*) has two references, *left* and *right*. These references point to a class that contains a list of elements from the source (*left*) or target (*right*) metamodels. *ListNotFound* contains a list with all the elements not found.

```
class NotFound extends WLink {
        reference left container : ListNotFound;
        reference right container : ListNotFound;
}
class ListNotFound extends WLink {
        -- @subsets end
        reference element [*] container : Element;
}
```

Fig. 11 Metamodel extension for elements not linked

6.4 Chains of matching transformations

In this section, we present a graphical approach to easily chain and to customize different matching transformations. As we have already seen, the matching transformations have a pre-defined signature. These transformations take a weaving model, a source model (or metamodel), and a target model (or metamodel) as input, and produce a new weaving model as output. The transformations are parameterized by values such as weight or threshold.

The parameterization of the matching transformations is defined in a configuration model. This configuration model contains parameters. This model conforms to a match parameter metamodel. This metamodel specifies the transformations that are executed, the execution order, and a set of tuning parameters. This allows combining different matching methods. The match parameter metamodel is illustrated in Fig. 12.

The class *ParameterSet* contains a set of transformations (ordered) and the set of metamodel extensions. The class *Transformation* defines the standard attributes of every transformation. A transformation is executed if the *selected* attribute is set to *true*. The reference *metamodels* contains the matching metamodels that need to be loaded to be able to execute a transformation. The reference *depends* indicates that one transformation can be executed only if a depending transformation has been previously executed. For instance, the similarity flooding transformation cannot be executed if the weaving propagation model has not been previously created. There are four kinds of transformations. They correspond to the different kinds of matching transformations presented in this section: *LinkGeneration*, *ElementToElement*, *Structural*, and *Filter*.

We define an interface that interprets the models conforming to this metamodel and produces a generic configuration window, as illustrated in Fig. 13. This interface is an example to illustrate the parameter model.

The configuration window has one group for each different kind of transformation. Each group shows the set of available parameters. The "?" (question mark) button shows the dependencies between the transformations and the metamodels. The "Save intermediate models" button saves a new weaving model after the execution of each matching transformation. This enables to compare the intermediate results. The configuration model loaded in this window is only illustrative (different values are used and explained in the validation

Fig. 12 Match parameter metamodel

```
package match_parameter {
        abstract class NamedElement {
                attribute name : String;
        }
        class ParameterSet extends NamedElement {
                reference transformations[*] ordered container: Transformation
                reference metamodels [*] container : Metamodel;
        }
        abstract class Transformation extends NamedElement {
                reference metamodels [*] : Metamodel;
                attribute description : String;
                attribute selected : Boolean;
                reference depends [*] : Transformation;
        }
        class LinkGeneration extends Transformation { }
        class ElementToElement extends Transformation {
                attribute weight : Double;
        }
        class Structural extends Transformation { }
        class Filter extends Transformation {
                attribute threshold : Double;
        }
        class Metamodel extends NamedElement {
                attribute description : String;
        }
}
```

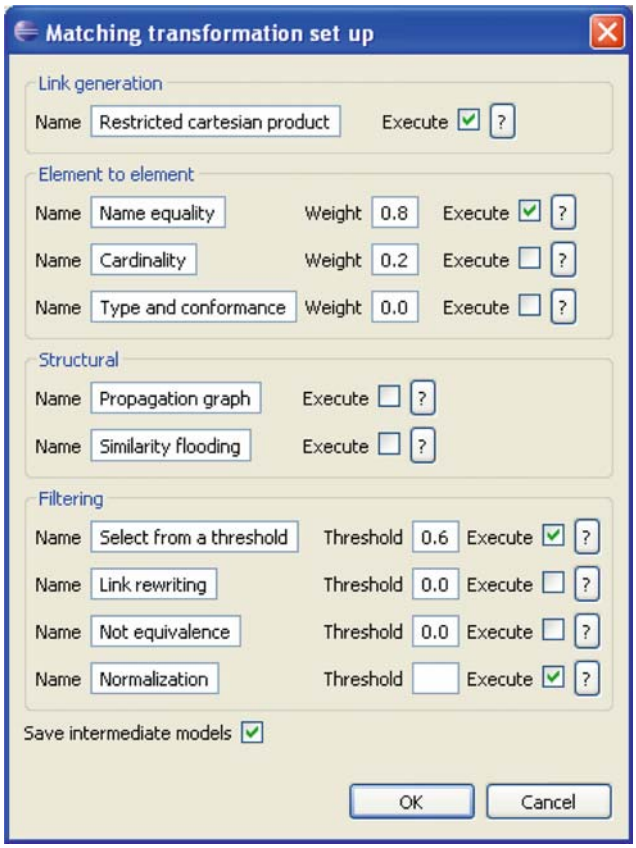

Fig. 13 Matching transformation configuration

section). The model defines the execution of the following transformations: a restricted Cartesian product based on the type of elements; a comparison over the element names, with a weight of 0.8; a comparison over the elements cardinality, with weight 0.2; a selection of the links with similarity value higher than 0.6; and the normalization of the results.

The configuration metamodel and model brings some advantages when combining the execution of different matching transformations. The correct tuning of matching transformations requires a lot of experience on matching. The configuration model enables recording these matching parameters. The model can be reused later by other developers. The tuning of matching transformations is a subject of study by itself. Each application scenario can have different parameters with the objective to obtain the best matching results. The variation of a single parameter can modify the final generated weaving model. Consequently, the exchange of these configuration models among the developers may help to better tune their methods and environments. An environment where these models can be easily reused is a significant help.

6.5 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. The higher-order transformations are based on a generic transformation pattern presented in [10].

The definition of the generic pattern of transformation relies on three facts. First, the core weaving metamodel is formed by links, link endpoints and extensions of these

Fig. 14 A HOT example

```
rule rule AttributeBinding {
  from
     amw : AMW!AttributeEqual
  to
    atl : ATL!Binding (
     propertyName <-
        MOF!EClassifier.getInstanceById(amw.target.element.ref).name,
     value <- amw.source
  )
}
```

elements. Second, declarative transformation languages have similar structure. Third, we use declarative patterns of transformation that specify only what to transform, and not how to transform. The pattern of transformation expresses the execution semantics of the weaving model, because it transforms the different kinds of links into executable mapping expressions in some transformation language.

The generic pattern is specified using higher-order transformations (HOT). A HOT takes as input a weaving model conforming to an extension of the weaving metamodel and transforms it into a transformation model.

Definition 6.4 (Higher-order transformation) A higher-order transformation is a transformation $\mathrm{T}_{OUT} : MM_T = T_{HOT}(T_{IN} : MM_T)$, such that the input and/or the output models are transformation models. Higher-order transformations either take a transformation model as input, either produce a transformation model as output, or both.

This pattern is the basis to define a model management operation called *TransfGen*. We define this operation below.

Definition 6.5 (TransfGen operation) *TransfGen* is a higher-order transformation that takes a weaving model M_W as input and that produces a transformation model M_T as output. The weaving model conforms to a domain-specific weaving metamodel extension MM_W.

$$M_T : MM_T = TransfGen(M_W : MM_W).$$

This operation is used to produce the model integration transformation that transforms the input terminal models into the output terminal models. We show in Fig. 14 an example of a HOT written in ATL, which produces an ATL model.

It produces a transformation binding (i.e., the expression *targetAttribute* ← *variable.sourceAttribute*). The *propertyName* is the name of the target attribute. The *getInstanceById* helper is used to recuperate the target element; *value* is the source element.

7 Experiment

In this section, we present an experiment using a metamodel comparison and model integration application. It has the same goal as the motivating example, i.e., to translate a set of input models into a set of output models. Metamodels need to be compared for several reasons. One important reason is to discover the equivalent elements between two versions of a metamodel. The result of a comparison is used to migrate between the models conforming to these metamodels. First, we present a general overview of the experiments. Second, we analyze the results grouped by the different kinds of matching transformations. Finally, we present the results of the generation of transformations.

7.1 General overview

Consider two versions of a *Scade* metamodel, *Scade (v1)* and *Scade (v2)*. The metamodels conform to the Ecore metametamodel [14]. An Ecore metamodel may have several packages. A package has a set of classes; each class has a set of attributes and references. *Scade* is a standard for development of embedded software for the Aircraft Industry [12] (the metamodels used here are not the official versions of the *Scade* suite). We do not describe in detail the metamodel elements, because they have approximately 400 elements. In brief, the metamodels define embedded components; the components are in a specific state; the state may change (using state machine concepts); the changes are specified using equations and arithmetic expressions. The $v2$ of the metamodel is derived from $v1$. This means the metamodels are semantically close. An organization using *Scade* decides to migrate from a model conforming to *Scade (v1)* into a model conforming to *Scade (v2)*.

We define two execution settings. In *setting 1*, we execute a set of matching transformations between two complete *Scade* metamodels. In *setting 2*, we extract one part of both metamodels. This enables the comparison of the execution of the matching transformations over metamodels with different size.

The number of elements of the metamodels of setting 1 is shown in Table 1. This table also shows the total number of elements, the number of classes, attributes and references. The sum of these three kinds of elements is less than the total because the metamodels have other kinds of elements not described here, such as data types.

Table 1 Number of elements of setting 1

	Elements	Classes	Attributes	References
Version 1	449	106	105	231
Version 2	381	95	89	190

Table 2 Number of elements of setting 2

	Elements	Classes	Attributes	References
Version 1	134	38	39	50
Version 2	114	28	49	30

The number of elements of the metamodels of setting 2 is shown in Table 2. The elements of these metamodels are extracted randomly. The metamodels have approximately 4 times less elements (1/4) than in setting 1.

We produce weaving models that contain links with equivalence semantics between the metamodels' elements. The comparison weaving model conforms to a weaving metamodel that is an extension of the core weaving metamodel. We use the extensions presented in the previous sections (e.g., $\langle Type \rangle$ *Equal*) to represent the equivalence between elements. *NotFound* links are used to store the elements that do not have any equivalence in the two metamodel versions.

In order to compare the results between both settings, we execute the same set of matching transformations, with the same input parameters. The execution parameters are saved in a configuration model. We execute 10 different matching transformations:

(1) Restricted Cartesian product.
(2) Name similarity: weight: 0.8.
(3) Cardinality similarity: weight: 0.1.
(4) Type and conformance similarity: weight: 0.1.
(5) Propagation graph with inheritance and relationship propagation.
(6) Similarity flooding.
(7) Threshold selection: value: 0.6.
(8) Select left best: only the links that have the best value for a given left element are selected.
(9) Link rewriting: the links are rewritten to represent the nested and inheritance relationships.
(10) Storage of links without equivalences.

The values of the parameters are chosen after executing the transformations several times under smaller examples (e.g., the motivating example), and then copied into our experiments. In the next subsections, we analyze the results grouped by the kind of matching transformation.

Table 3 Links created by a restricted Cartesian product

	Total links	Class links	Attribute links	Reference links
Setting 1	63.305	10.070	9.345	43.890
Setting 2	4.684	1.064	1.170	2.450

Table 4 Links with the same name

	Class links	Attribute links	Reference links
Setting 1	95	437	593
Setting 2	28	36	49

7.1.1 Weaving creation

Table 3 shows the number of links created in both settings. The restricted Cartesian product enables to diminish the number of possible links, especially in setting 1. The total number of links is 36% smaller. The Cartesian product would produce $449 \times 381 = 171.069$ links. The same is valid for setting 2, where the Cartesian product would produce $134 \times 114 = 15.276$ links. The resulting weaving model has 31% of the number of possible links.

7.1.2 Element-to-element similarities

Table 4 shows the number of links that have the same name, thus, with similarity equal to 0.8.

Although the elements with the same name have a high probability to be similar, this does not guarantee that the links are all correct. This is the case of attributes and references, because different classes may contain attributes with the same name. The most common case is the attribute *name*, which is present in a large number of classes. The matching transformation that interprets the cardinality values enables to increase the similarity of references. This transformation does not impact the similarity of attributes, because attributes have cardinality [1–1]. This makes a more accurate distinction between different references with the same name. The type and conformance transformation increase the similarities between attributes and references.

Both settings have similar results with respect to the proportion between elements with high similarity values. However, the big difference on the number of total links has a major impact on the overall performance, because the transformations visit every pair of links to compute the similarity estimation. The average number of elements of setting 1 is only 4 times bigger than in setting 2. However, the restricted Cartesian product creates 13.5 times more links.

We evaluate the performance of these transformations using a Pentium 4, at 2.6 MHz, with 1GB of RAM. Table 5

Table 5 Execution time of both settings

	Sequential	Combined
Setting 1	1.320	360.73
Setting 2	50.87	14.87

shows the execution time, in seconds, of both settings. First, we execute the matching transformations separately, one by one (sequential approach). This includes model loading, saving, and computation of similarities. The result of each transformation is saved in a separated weaving model. Second, we execute a single transformation that combines all the methods (combined approach) and produces only one weaving model with the final results.

In setting 1, the combined approach is executed 3.6 times faster than the sequential approach. In setting 2, the combined approach is 3.4 faster. Although the combined approach has better performance, it is less modular, because the methods are manually coded in a single transformation. This makes it difficult to reuse and parameterize the methods for different scenarios. Moreover, the sequential approach enables comparing the intermediate results of each transformation.

In the sequential approach, setting 2 is executed 25.9 times faster than setting 1. In the combined approach, setting 2 is executed 24.2 times faster than setting 1. The performance ratio between both settings is almost the same in both executions.

This high performance gain motivates the execution of the matching transformations using smaller metamodels. This means we may execute the matching transformations much faster if we separate the metamodels of setting 1 in 4 submetamodels. However, it is still necessary to check if the weaving models produced have the same links.

7.1.3 Structural similarities

After the element-to-element similarities, we create a weaving propagation model and execute the similarity flooding transformation. These transformations enable producing a more accurate result, because they increase the similarity between references or attributes.

The main advantage of these methods is to increase the similarity by analyzing the containment and inheritance relations between the metamodel elements. These methods increase the difference between the elements with higher and lower similarities. This enables better distinguishing element equivalencies. For instance, the average difference between the lowest and the highest similarity values is 0.2 using element-to-element transformations. Using the propagation graph and the similarity flooding, the difference augmented to 0.6.

Table 6 Number of links of the link rewriting transformation

	Class links	Attribute links	Reference links
Setting 1	95	91	193
Setting 2	28	29	31

Table 7 Elements not found and precision

	Not found	Precision (%)
Setting 1	95	91
Setting 2	28	73

7.1.4 Selection of best links

The last kind of transformations is the selection of the best links. The threshold of 0.6 filters most of the links. This threshold value does not have an impact on the final weaving model, because the subsequent selection methods execute more refined solutions. However, it improves performance, since the subsequent methods have less links to process. The threshold transformation returns a number of links that is approximately the same as the links with a high similarity of name.

The select left best and nested rewriting methods produce a weaving model very close to the final result. Table 6 shows the number of links created.

We can see that the biggest difference with the element-to-element transformations is on the number of links between attributes and references.

All the attribute and reference links are rewritten as children of a class link. In addition, setting 1 creates 71 class links that are child of other class links. This means that the transformations reconstruct the inheritance relationships between classes. The last link rewriting transformation produces *NotFound* links to store the elements without equivalence links. Table 7 shows the number of elements not found in both settings and the precision of the transformations (the precision is the percentage of correct links).

The precision of setting 1 is better than setting 2. The main reason is because setting 1 has all the metamodel elements to compute the similarities. The metamodel excerpts of setting 2 were created randomly. Consequently, the transformations have less information to compute the similarities. However, setting 2 has much better performance results. The choice of the appropriate method depends on the precision desired, as well as on the computational resources available. In addition, the matching of metamodels with too many elements hardens the task of verifying the correctness of the results.

```
abstract rule Object_2_Object26 {
  from
    v_left : Scadev1!Object
  to
    v_right : Scadev2!Object (
      name <- v_left.name,
      runLine <- v_left.runLine
    )
}
rule Label_2_Label21 extends Object_2_Object26 {
  from
    v_left : Scadev1!Label
  to
    v_right : Scadev2!Label (
      expression <- Set {v_left.expression}
    )
}
```

Fig. 15 A part of the transformation generated automatically

7.1.5 Transformation generation

Once the weaving model is created, it is interpreted by a HOT that transforms the declarative links of the weaving model into an executable ATL transformation. Each *Equal* link is transformed into an ATL rule. The left and right references are transformed into the input and output elements. The *AttributeEqual* and *ReferenceEqual* links are transformed into bindings.

Figure 15 shows an excerpt of the transformation that is generated for setting 1.

This transformation contains binding expressions between the equivalent elements. The identifiers after the rule names are automatically generated. This model transformation is responsible to translate the model conforming to *Scade (v1)* into the model conforming to *Scade (v2)*. The transformation generated in setting 1 has 1030 lines and that in setting 2 has 282 lines.

7.1.6 Summary

This experiment demonstrates how weaving models are used to compare different metamodels and to migrate between the models conforming to these metamodels. It shows that the matching transformation scales to large metamodels. However, better performance results are achieved with smaller metamodels, nevertheless, with lower precision. Thus, the choice of the appropriate transformation depends on the precision desired, as well as on the resources available.

The different kinds of links enable to clearly identify the relationships between the elements. The resulting weaving model can be refined using graphical facilities of the weaving engine used. The user interface of the weaving engine helps a lot, because we are not experts in the aircraft domain.

We choose to implement a set of simple methods between two metamodels that are semantically close. In order to achieve good results with metamodels with a larger number of differences, additional methods should also be implemented. For instance, domain ontologies could be used to improve the result.

The easy configuration of the matching transformations enables using methods adapted to the comparison use case. The weaving extensions are simple, and can be generalized for different scenarios. Finally, the weaving model is used to successfully generate the executable ATL transformation to perform the model migration.

The experiment is developed using the AMMA platform (ATLAS Model Management Architecture). AMMA provides open-source plug-ins for model transformation (ATLAS Transformation Language [21]) and for model weaving (ATLAS Model Weaver [9]).This use case is available for download at (http://www.eclipse.org/gmt/amw/usecases/compare/). This page contains a fully implemented example, with general documentation, a *HowTo* and the sources.

8 Related work

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 [2,8,24], ontology merging and alignment [15,29,30], or ontology integration [13]. Among these several approaches, COMA++ [2] and the API of Euzenat [15] are the solutions most similar to ours.

COMA++ implements a set of methods, using, for instance, element-to-element methods or incremental matching. COMA++ provides an interactive user interface to combine these methods. 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. Other methods can be implemented and integrated into our platform, for instance, to use domain ontologies to help on the matching process.

The work of Euzenat [15] factors out schema matching features and proposes a generic API. The API provides interface methods used to implement different matching methods 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 methods to match different metamodels. However, we do not provide operations to evaluate different matching results.

The solution in [13] proposes machine learning techniques to select among a set of methods, and not to create methods as in most part of solutions. This is a complementary approach.

The machine learning techniques could be enhanced to support our matching transformations as input.

iMAP [8] is one of the few approaches that create 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.

The weaving models and metamodels are similar to triple graph grammar. The major differences are that weaving models conform an explicit metamodel, and they can also be transformed. This enabled the use of weaving models as input to the matching transformations.

The work from [34] presents a solution to help on the creation of model transformations. The main difference is that it uses mappings between terminal models, and not metamodels. MTBE [35] also uses mappings between terminal models. The model transformation is then derived from these mappings. These approaches can be effective when using terminal models with reasonable size. However, they must have at least one sample of the input and output models, which is not the case of our approach.

Similarity Flooding (SF) [25] is a generic structural method that propagates the similarity of a pair of nodes through their neighbors. However, as the authors say, 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 method to support the matching of different metamodels, with two different ways to propagate the similarities through the neighbors' elements.

Clio [28] is one of the first solutions to provide a semi-automatic mechanism to produce transformations based on a set of relationships. Our proposal has a similar architecture. Clio has evolved in [17] to support the production of more complex mappings with nested structures. However, the definition of the relationships cannot be extended, which hardens the task of creating complex kinds of mappings. Furthermore, Clio does not provide facilities to easily develop or adapt matching methods, which are coded in some programming language and plugged into the tool.

Model management solutions [3,5,25,27] propose to create operations that encapsulate the most frequently executed metadata tasks. The transformation and weaving engines presented in our approach can be used as a generic solution to develop other model management operations.

Model management solutions have been initially developed to consider simple kinds of mappings. The work in [6] revises such solutions stating that model management must support operations over complex mappings and that the runtime is a central component. This tightens the relation between model management and our solution, since the two key aspects of our approach is the extensibility of the weaving metamodels and the execution of different operations using model transformations.

The work from [26] describes a mapping compilation solution to bridge between applications and databases. Mapping compilation is an important aspect that has several research issues yet to be solved. The mapping compilation phase is equivalent to the generation of transformations of our solution and Clio. The authors exploit different aspects specific to databases, such as outer joins and case statements.

The solution in [31] enables automating the tuning of matching parameters using synthetic scenarios. The results of such solution could be stored in the form of our configuration model to be shared among developers.

9 Conclusions

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

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

The matching transformations enable the creation of weaving models between models conforming to different metamodels, and the creation of links only between a restricted set of elements. We proposed a metamodel-based matching transformation that enables propagating the similarity between elements in different ways. We presented a link rewriting operation that analyzes the relationships between the links of a weaving model. These links are transformed into complex kind of links. In addition, we presented a transformation that saves information about the elements that are not matched, enabling to easily identify them.

We proposed saving the execution parameters of the matching transformations in a configuration model. This model is used to share these execution parameters. This is an important achievement, since the tuning of matching transformations is a difficult task.

To validate our approach, we conducted a set of experiments. The matching of large metamodels had better precision for the example provided. This does not mean, however,

that small metamodels cannot produce good results. Matching large metamodels may have performance problems due to the large number of links. Moreover, too large metamodels make it difficult to verify if the results are correct.

The optimization of these matching transformations 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 can be merged by a transformation engine to be executed in a single rule. The major performance concern is caused by the number of links created by the Cartesian product. There are different possibilities to diminish the number of initial links. For instance, to filter the links already on the moment of creation, by creating links only with similarity higher than a minimum bound. We saw that the development of matching transformations helps on the specification of methods because it diminishes the gap between the conceptual structures and the implementation. This motivates the creation of a domain specific language for developing matching methods, facilitating even more the development, testing and reusability of such methods.

Acknowledgments This work is partially supported by ModelPlex European integrated project FP6-IP 034081 (Modeling Solutions for Complex Systems).

Appendix A

This appendix presents the links created and the transformation produced for the motivating example of Sect. 2.

We execute the following matching transformations (in that order): (1) Cartesian product, (2) string comparison—Levenshtein distance (with weight 1.0), (3) propagation weaving model, (4) similarity flooding, (5) link rewriting and (6) link rewriting (abstract refactoring). The transformation 1 creates the set of links. The transformations 2, 3 and 4 modify the similarity value, without adding or deleting any links. The transformations 5 and 6 select the links with higher similarity value and re-organize the links in the weaving models. We show the results of these 2 last transformations below.

The link rewriting transformation produces the links shown in Table 8. These links are saved in a weaving model.

After the first rewriting of links, there are still some adjustments to be done. It is necessary to reorganize the links involving abstract classes (*Person*), because abstract classes are not transformed. Table 9 shows the links created after executing the refactoring transformation.

The transformation deletes the *ElementMatch* links that have at least one endpoint referring to an abstract class. However, it does not delete the *AttributeBinding* links, since they are needed to produce the final transformation.

Table 8 Link rewriting transformation results

Type of link	Source	Target
ElementMatch	Person	Professor
AttributeBinding	Person : name	Professor : name
AttributeBinding	Person : SSN	Professor : SSN
AttributeBinding	Person : city	Address : city
AttributeBinding	Person : zip_code	Address : code
AttributeBinding	Person : street	Address : street
ElementMatch	Teacher	Professor
ElementMatch	Person	Student
AttributeBinding	Person : name	Student : name
AttributeBinding	Person : SSN	Student : SSN
ElementMatch	Student	Student
ElementMatch	Undergraduate	Student
ElementMatch	Master	Student
ReferenceBinding	Master : advisor	Student : advisor

Table 9 Link rewriting (abstract refactoring) results

Type of link	**Source**	**Target**
AttributeBinding	Person : name	Professor : name
AttributeBinding	Person : SSN	Professor : SSN
AttributeBinding	Person : city	Address : city
AttributeBinding	Person : zip_code	Address : code
AttributeBinding	Person : street	Address : street
ElementMatch	Teacher	Professor
AttributeBinding	Person : name	Student : name
AttributeBinding	Person : SSN	Student : SSN
ElementMatch	Undergraduate	Student
ElementMatch	Master	Student
ReferenceBinding	Master : advisor	Student : advisor

These links are used to produce the transformation shown in Listing A.1. The *Inschema* represents the input metamodel (*MM1*), and the *Outschema* represents the output metamodel (*MM2*). The transformation has three rules; each rule has a set of bindings. The transformation copies the bindings of the abstract class *Person* to its children classes. This transformation is automatically produced. However, there is one manual refinement that must be done to obtain the final desired transformation (the same transformation of the motivating example). We manually create a new element *Address* for each rule, and we correct the "*address*" bindings. The use of matching transformations enables obtaining a good first transformation, with some minor refinements. This avoids writing repetitive transformation code.

Listing A.1. Transformation produced automatically

```
module T_SimplePersonExample;

create OUT : Outschema from IN : Inschema;

rule Teacher_2_Professor {
  from
    v_left : Inschema!Teacher
  to
    v_right : Outschema!Professor (
      name <- v_left.name
      SSN <- v_left.SSN,
      -- REFINE: handle the creation of a new element
      street <- v_left.address.street,
      city <- v_left.address.city,
      code <- v_left.address.zip_code,
}

rule Undergraduate_2_Student {
  from
    v_left : Inschema!Undergraduate
  to
    v_right : Outschema!Student
      name <- v_left.name
      SSN <- v_left.SSN,
      -- REFINE: handle the creation of a new element
      street <- v_left.address.street,
      city <- v_left.address.city,
      code <- v_left.address.zip_code,
}

rule Master_2_Student {
  from
    v_left : Inschema!Master
  to
    v_right : Outschema!Student (
      advisor <- v_left.advisor
      name <- v_left.name
      SSN <- v_left.SSN,
      -- REFINE: handle the creation of a new element
      street <- v_left.address.street,
      city <- v_left.address.city,
      code <- v_left.address.zip_code,
    )
}
```

References

1. AM3 Atlantic Zoo. http://www.eclipse.org/gmt/am3/zoos/atlanticZoo/. Jun. 2007
2. Aumueller, D., Do, H.H., Massmann, S., Rahm, E.: Schema and ontology matching with COMA++. In: Proceedings of SIGMOD 2005, pp. 906–908
3. Atzeni, P., Cappellari, P., Bernstein, P.A.: Model independent schema and data translation. In: Proceedings of EDBT 2006, pp. 368–385
4. Balogh, A., Németh, A., Schmidt, A., Ráth, I., Vágó, D., Varró, D., Patricza, A.: The VIATRA2 model transformation framework. In: Proceedings of ECMDA 2005—Tools Track, 2005
5. Bernstein, P.A.: Applying model management to classical meta data problems. In: Proceedings of CIDR 2003, pp. 209–220
6. Bernstein, P.A., Melnik, S.: Model management 2.0: manipulating richer mappings. In: Proceedings of SIGMOD 2007. Beijing, China, pp. 1–12
7. Cohen, W., Ravikumar, P., Fienberg, S.E.: A comparison of string distance metrics for name-matching tasks. In: Proceedings of IIWeb 2003, pp. 73–78
8. Dhamanka, R., Lee Y., Doan, A., Halevy, A., Domingos, P.: iMAP: discovering complex semantic matches between database schemas. In: Proceedings of SIGMOD 2004, pp. 383–394
9. Didonet Del Fabro, M.: Metadata management using model weaving and model transformations. Ph.D. thesis, University of Nantes, September 2007
10. Didonet Del Fabro, M., Bézivin, J., Valduriez, P.: Model-driven tool interoperability: An application in bug tracking. In: Proceedings of ODBASE'06, LNCS 4275, Nov. 2006, pp. 863–881
11. Didonet Del Fabro, M., Valduriez, P.: Semi-automatic model integration using matching transformations and weaving models. In: Proceedings of the 22nd Annual ACM SAC, MT 2007—Model Transformation Track, Seoul (Korea), pp. 963–970
12. Dion, B.: Efficient Development of Safe Railway Applications Software with EN 50128 Objectives using SCADE Suite, Innotrans 2006
13. Ehrig, M., Staab, S., Sure, Y.: Bootstrapping ontology alignment methods with APFEL. In: Proceedings of the 4th ISWC 2005, Galway, Ireland, volume 3729 of LNCS, pp. 186–200
14. EMF. Eclipse Modeling Framework. http://www.eclipse.org/emf
15. Euzenat, J.: An API for ontology alignment. In: Proceedings of ISWC 2004, pp. 698–712
16. Fellbaum, C. WordNet, an Electronic Lexical Database. MIT Press, Cambridge (1998). http://wordnet.princeton.edu/
17. Fuxman, A., Hernández, M.A., Ho, H., Miller, R.J., Papotti, P., Popa, L.: Nested mappings: Schema mapping reloaded. In: Proceedings of VLDB 2006, Seoul, Korea, pp. 67–78
18. 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
19. Jouault, F., Bézivin, J.: KM3: a DSL for metamodel specification. In: Proceedings of 8th FMOODS, LNCS 4037, Bologna, Italy, 2006, pp. 171–185
20. Jouault, F., Kurtev, I.: On the architectural alignment of ATL and QVT. In: Proceedings of the 2006 ACM Symposium on Applied Computing (SAC 06). ACM Press, Dijon, France, 2006, chapter Model transformation, pp. 1188–1195
21. Jouault, F., Kurtev, I.: Transforming models with ATL. In: Proceedings of the Model Transformations in Practice Workshop at MoDELS 2005, Montego Bay, Jamaica, pp. 128–138
22. JWNL (Java WordNet Library). http://sourceforge.net/projects/jwordnet. August 2006
23. Königs, A.: Model Transformation with Triple Graph Grammars, Model Transformations in Practice Satellite Workshop of MODELS 2005, Montego Bay, Jamaica (2005)
24. Madhavan, J., Bernstein, P.A., Rahm, E.: Generic schema matching using cupid. In: Proceedings of VLDB 2001, pp. 49–58
25. Melnik, S.: Generic Model Management: Concepts and Algorithms. Ph.D. Dissertation, University of Leipzig, Springer LNCS 2967 (2004)
26. Melnik, S., Adya, A., Bernstein, P.A.: Compiling mappings to bridge applications and databases. In: Proceedings of SIGMOD 2007. Beijing, China, pp. 461–472
27. Melnik, S., Bernstein, P.A., Halevy, A., Rahm, E.: Supporting executable mappings in model management. In: Proceedings of SIGMOD 2005, Maryland, US, pp. 167–178
28. Miller, R.J., Hernandez, M.A., Haas, L.M., Yan, L.-L., Ho, C.T.H., Fagin, R., Popa, L.: The Clio Project: Managing heterogeneity. In: SIGMOD Record 30, 1, 2001, pp. 78–83
29. Mitra, P., Wiederhold, G., Kersten, M.: A graph-oriented model for articulation of ontology interdependencies. LNCS, 1777:86+ (2000)
30. Noy, N., Musen, M.: PROMPT: Algorithm and tool for automated ontology merging and alignment. In: Proceedings of. AAAI/IAAI, pp. 450–455

31. Sayyadian, M., Lee, Y., Doan, A.H., Rosenthal, A.: Tuning schema matching software using synthetic scenarios. In: Proceedings of VLDB 2005, Trodheim, Norwary, pp. 994–1005
32. SimMetrics. Developed by Sam Chapman. http://sourceforge.net/~projects/simmetrics/. August 2006
33. Shvaiko, P., Euzenat, J.: A Survey of Schema-Based Matching Approaches. JoDS IV, pp. 146–171 (2005)
34. Varró, D.: Model transformation by example. In: Proceedings of MoDELS/UML 2006, Genova, Italy, pp. 410–424 (2006)
35. Wimmer, M., Strommer, M., Kargl, H., Kramler, G.: Towards model transformation generation by-example. In: Proceedings of 40th HICSS 2007, CD-ROM / Abstracts Proceedings, 3–6 Jan. 2007, Waikoloa, Big Island, HI, USA

Author Biographies

Marcos Didonet Del Fabro is a post-doc at ILOG SA, in collaboration with INRIA labs, in France. He received his Ph. D. degree in Computer Science from the University of Nantes in 2007. He is responsible of Eclipse/GMT component AMW (ATLAS Model Weaver), and also a contributor to AM3 and ATL components. He has worked 7 years as a software developer in Brazil. His current research is about different use cases of model weaving and model transformations, applied to Business Rule Management Systems.

Patrick Valduriez is a Director of Research at INRIA, France, and the manager of the Atlas research team in Nantes pursuing research in data management in distributed systems. He received his Ph. D. degree and Doctorat d'Etat in Computer Science from the University Paris 6 in 1981 and 1985, respectively. He has authored and co-authored over 170 technical papers and several books, among which, "Principles of Distributed Database Systems" published by Prentice Hall in 1991 and 1999, "Object Technology" published by Thomson Computer Press in 1997, and "Relational Databases and Knowledge Bases" published by Addison-Wesley in 1990. He has been a trustee of the VLDB endowment and an associate editor of several journals, including ACM Transactions on Database Systems, the VLDB Journal, Distributed and Parallel Databases, Internet and Databases, Web Information Systems, etc. He has been general chair of SIGMOD/PODS'04 in Paris and of EDBT'08 in Nantes. In 2009, he is the general chair of VLDB in Lyon. In 1993, he was the recipient of the IBM scientific prize in France. He obtained the best paper award at the VLDB'00 conference.