AMW: a tool for multi-level specification of model transformations

Marcos Didonet Del Fabro[a,*], Frédéric Jouault[b], Jordi Cabot[b], Patrick Valduriez[c]

[a] *Univ. Federal do Paraná, Depto. de Informática, Centro Politécnico, Curitiba, PR, Brazil*
[b] *AtlanMod INRIA Team, EMN, Nantes, France*
[c] *INRIA - LIRMM, Montpellier, France*

Abstract

Model transformations play a key role in any Model Driven Engineering (MDE) approach. The central task when developing model transformations consists in writing rules that implement a set of mappings between two metamodels. Specification of model transformations is far from being a simple task. For complex transformations, attempting to directly write the transformation is an error-prone and time-consuming process. To simplify the development of model transformations, we present a multi-level approach inspired on typical software development processes where we first create higher level and abstract mappings' specifications and then we refine them into more precise transformation code. To support this multi-level transformation approach we have developed the AMW (AtlanMod Model Weaver) tool. AMW facilitates the specification of transformations in a three-level setting, from high-level mapping specifications to the generation of the final transformation code. A set of extension points allow the configuration and adaptation of each refinement level to the specific requirements of each application scenario, if desired. We present the tool capabilities using two different settings: manual and semi-automatic transformation generation.

Keywords: model transformations, model weaving, transformation specification, AMW

1. Introduction

Model Driven Engineering (MDE) is a software engineering paradigm that emphasizes the use of models in all software engineering activities. Therefore,

*Corresponding author
Email addresses: marcos.ddf@inf.ufpr.br (Marcos Didonet Del Fabro), frederic.jouault@inria.fr (Frédéric Jouault), jordi.cabot@inria.fr (Jordi Cabot), patrick.valduriez@inria.fr (Patrick Valduriez)

model transformations (for simulation, code-generation, reverse engineering, testing and so on) are amongst the most used operations in MDE applications.

The central development task when writing model transformations consists in manually writing transformation rules, where each rule define a partial mapping between two metamodels. Most transformation languages only support the definition of unidirectional rules, i.e., rules that specify how a set of elements from an input model are translated into a set of elements of an output model. Examples of transformation languages are ATL [1], QVT [2], Viatra [3], or Kermeta [4] .

Unfortunately, developing and understanding large model transformations is a difficult task, because many rules may require complex pattern expressions to precisely define the mapping (e.g. a source pattern to filter the input model and a target pattern to explain how the elements from the input model are combined to create the elements of the target model).

In order to simplify the development of model transformations, we propose to adopt a three-level approach inspired by software development processes [5]. In the third level, we create abstract transformation specifications as a set of declarative element-to-element relationships. In the second level, these relationships are used as input to generate a partial model transformation (i.e., a skeleton) using a specific model transformation language (ATL in our case). In the last level, the developer is responsible for completing the transformation with the patterns that could not be deduced in the previuos step.

In this paper, we present AMW [1], an Eclipse-based [6] tool to support this multi-level specification of transformations. The abstract specifications (i.e., the declarative relationships) are created using a combination of manual definitions and heuristics that automatically estimate the similarity between the model elements. The relationships are stored in a model, called *weaving model* [7]. Then, the tool uses this weaving model for generating more specific (and lower-level) transformation code in the ATL language. The tool architecture follows the Eclipse paradigm of contributions: it is a generic workbench that provides a set of extensions points for tailoring, when needed, each level to the needs of a specific use case scenario. We present the tool capabilities using two approaches: manual and semi-automatic transformation generation. To the best of our knowledge AMW is the only tool providing this kind of support.

This article is organized as follows. Section 2 introduces basic MDE concepts. Section 3 presents a motivating example. Section 4 defines what is multi-level specification of transformations. Section 5 presents the AMW support for transformation specification and generation based on a guiding example. Section 6 describes how the manual process can be combining with heuristics for automatically discovering the relationships. Section 7 is the related work. Section 8 concludes.

[1]The tool web site, with download instructions and documentation is available at: http://www.inf.ufpr.br/didonet/amw/

2. Model Driven Engineering

The primary software artifacts of any MDE approach are models, which are considered as first-class entities. These models may be defined using Object Management Group (OMG) [8] standards (e.g., UML [9]) or de facto standards (e.g., Eclipse Modeling Framework (EMF) [10]).

A *model* represents a certain aspect of a system. Every model conforms to a metamodel, which defines its abstract syntax. A *metamodel* is a model that defines the type of the elements and relationships of a model. This relation is called *conformance*. In the same way a grammar in a programming language defines the possible programs that may written with it, a metamodel defines a set of constraints that all models must conform to. In its turn, a metamodel conforms to a metametamodel (EMF Ecore [10] and OMG MOF [11] are examples of metametamodels). A *metametamodel* is a model that specifies the base representation for all models and metamodels in a given context. A metametamodel conforms to itself .

Operations between different models are defined using model transformations. A *model transformation* is an operation that takes a set of models as input, visits the elements of these models and produces a set of models as output. There are several efforts that study model transformations; for instance, ATL [1], QVT [2], VIATRA [3], and others. These solutions have an heterogeneous set of characteristics: bidirectional $\times$ unidirectional; imperative $\times$ declarative $\times$ hybrid; support to traceability; incrementality; rule-based $\times$ code-based (a detailed classification is found at [12]).

Developing transformations is a common task in model-driven software scenarios. It is becoming complex and difficult to manage. However, there is no "transformation development method" that combines the benefits of these different languages in a single framework. This can facilitate the definition of transformation - and as we will see in the next section - presents some challenges. This means more effort is needed to improve the development of model transformations.

In a pure MDE approach, model transformations can also be regarded as models (conforming to a model transformation metamodel). Therefore, model transformations can be used as input and/or output of other model transformations (what we call Higher-Order Transformations, [13]) for instance, as part of a refinement process.

3. Motivating example

We illustrate the need for a three-level specification of model transformations using the "UML to Java" transformation. The translation of a UML class diagram (platform independent) into Java (platform specific) is a very common scenario when developing software applications. Consider, for instance, the Pet

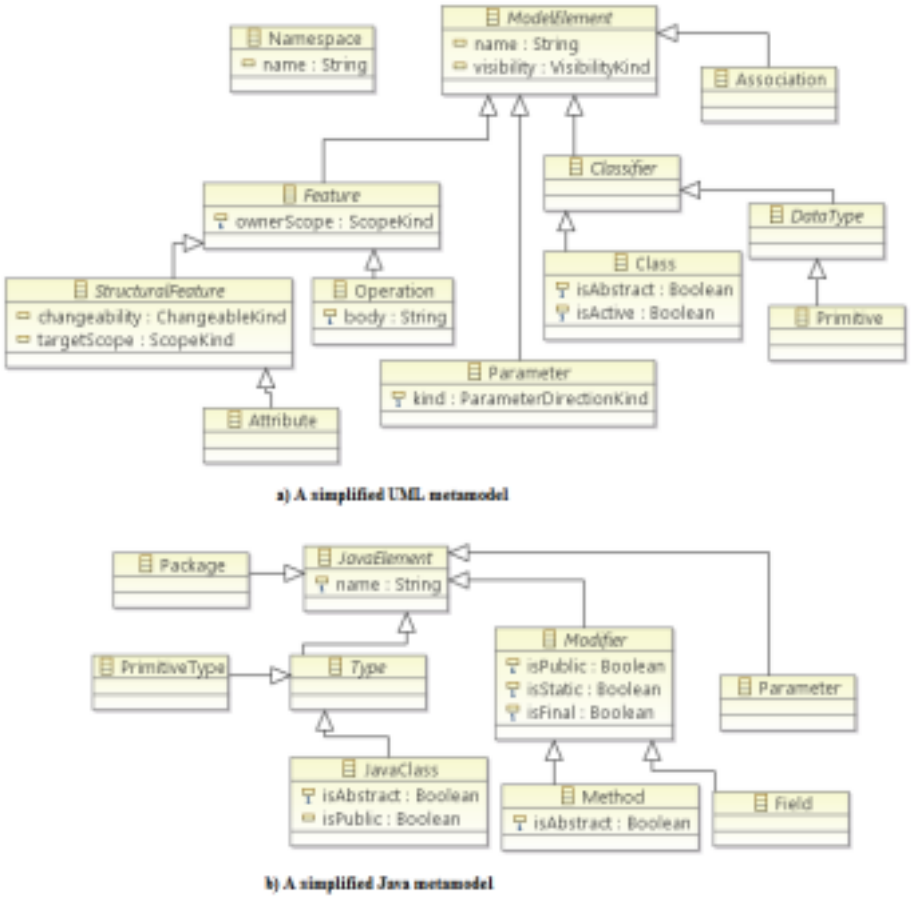

Figure 1: UML and Java metamodels. We show only the inheritance relationships for better visualization.

Store application [14] [2]. The Pet Store application can be defined in a UML model, with UML classes such as *Catalog*, *Item*, *Category*, *Product* and associations between them [3]. This example PetStore model conforms to a (simplified) UML metamodel shown in Figure 1 and must be translated to a Java model conforming to a (simplified) Java metamodel (Figure 1) [4].

This translation is implemented as a model to model transformation that takes as input a UML model and produces as output a Java model. The trans-

[2]The Pet Store is a reference application used by Oracle-Sun to illustrate J2EE applications

[3]We do not describe the Pet Store UML model here because it is not relevant for the definition of the transformation, which can be applied on any UML and Java models.

[4]Later on, this Java model can be translated into actual Java code by means of a model-to-text transformation but this is out of the scope of this paper

formation consists of a set of transformation rules where each rule is in charge of translating a (set of) UML element/s into a (set of) Java element/s as shown below. We use the ATL transformation language [1] to write the rules. We choose ATL because it's a mature language, with good tool support, a user community and it's one of the most used transformation language nowadays.

ATL transformations are unidirectional (from source to target). The basic construct in ATL are rules, which may be declarative or imperative (here we focus on declarative rules). A declarative rule is composed of a source and a target pattern (the *from* and *to* constructs, respectively). The source pattern is the type of the input element that is matched and then transformed. It may have a guard (i.e., a condition), which restricts the matched elements. The target pattern is the type of the output element that is created. It contains a set of attribute bindings, which define how to populate the attributes and references of the output element.

The first rule specifies that each *Class* of the input model is transformed into a *JavaClass* of the output model. It defines as well how the properties of the class are translated. The second rule is similar, but for transforming UML attributes into Java fields.

```
rule UMLClass2JavaClass {
   from e : UML!Class
   to out : JAVA!JavaClass (
      name <- e.name,
      isAbstract <- e.isAbstract,
      fields <- e.ownedElement->select
       (element | element.oclIsKindOf(IN!Attribute)),
      isPublic <- (e.visibility = #vk_public),
      package <- e.namespace
)
}
rule Attr2Field {
   from e : UML!Attribute
   to out : JAVA!Field (
      name <- e.name,
      isStatic <- (e.ownerScope = #sk_static),
      isPublic <- (e.visibility = #vk_public),
      isFinal <- (e.changeability = #ck_frozen),
      owner <- e.owner,
      type <- e.type
)
}
```

The ATL IDE [15] enables to develop the whole transformation manually, i.e., in a single-level setting. However, a single-level approach is not always the best solution, because the development of these rules requires a detailed comprehension of the transformation language, and of the input and output metamodels. A direct writing of all the patterns that may appear in the transformation

rule is very time-consuming and error-prone (e.g. the attribute bindings may become very complex). We can easily observe this complexity in several scenarios just by browsing through the ATL transformation repository available at [16].

On early phases, developers may not want to create the transformation with all the rules complete and operational, but may want to have only a general overview of the transformation where only some of the mappings are expressed and in high-level view. For instance, the binding of the *field* reference contains an expression that navigates through all elements in the *e.ownedElement* reference and it selects only *Attribute*'s. We may want to create a direct link between *fields* and *Attribute*, and refine the transformation latter on.

In such cases, we should provide a different transformation specification language, in the form of element-to-element mapping relationships, preferably using graphical interfaces. This will help designers to explore different transformation alternatives.

Only in subsequent phases, when the transformation scenario is clear, this initial specification should be refined and used to produce the final executable transformation.

4. Multi-level specification of model transformations

As we have discussed in the previous section, a direct (i.e. single-level) transformation strategy may not the best option when dealing with complex transformation scenarios. Similarly to existing software development processes, design of complex transformations should follow an iterative refinement process moving from a high-level transformation specification to a low-level transformation code. This is what we call *multi-level specification of model transformations*.

Multi-level specification of model transformations consists in a set of steps MT that produces a set of model transformations $MT = \{MT_L, MT_{L-1}, ..., MT_1\}$, $L = [N..1]$, such that:

- MT_L is the transformation with the highest abstraction level.
- for each level $L = [N..2]$, there is a function $\Phi_{(L,L-1)} : MT_L \rightarrow MT_{L-1}$. This means MT_L is used as input to produce MT_{L-1}.
- MT_1 is the final executable transformation that is (optionally) manually refined.

The number of levels can be arbitrary, however, we recommend to adopt a three-level approach :

- Abstract representation of transformations. For instance, graphically creating the relationship $Class \leftrightarrow JavaClass$.
- Incomplete transformation generation from the relationships information. For instance, producing a skeleton rule $UMLClass2JavaClass$ with the *from* and *to* parts and one-to-one links between the attributes (e.g., $fields \leftrightarrow e.ownedElement$).

- Manually-refined transformation from the skeleton. For instance, adding the equality expressions for the attributes and adding the navigation and selection expression for the $fields \leftrightarrow e.ownedElement$ relationship.

In the following sections we explain the AMW capabilities for supporting these three levels, using the UML2Java application as illustrative example. Note that the illustrations present excerpts of the generated artifacts, due to space restrictions. The full implementation can be found on the AMW site.

5. Multi-level specification of model transformations with AMW

AMW implementation of the previous three level transformation specification can be summarized as follows:

- *Level 3*: Abstract mapping relationships are created, modified and stored in a weaving model using the AMW graphical interface (creation, modification, storage).
- *Level 2*: Transformation skeletons targeting the ATL language are generated from the weaving model.
- *Level 1*: Generated ATL transformations are manually refined using the ATL IDE.

AMW produces the models for levels 3 and 2. Level 1 uses the ATL IDE. In the following sections, we explain the AMW support (concepts and tooling) for levels 3 and 2, respectively.

5.1. Abstract specification of transformations : level 3

We use weaving models for capturing the relationships between model elements. In this section, first we explain the weaving model concepts. Second, we explain how to initialize weaving models using the AMW tool. Third, we explain how to edit these models.

5.1.1. Weaving model concepts

AMW is based on the concept of weaving model. A weaving model is a model that stores relationships (i.e., links) between elements belonging to different models. A weaving model conforms to a weaving metamodel. The weaving metamodel defines the kinds of links that may be created between elements as shown in Figure 2.

The core metamodel has metaclasses to specify information about links types, their identification and the elements they link to. The weaving model conforming to this metamodel will contain the actual links.

More specifically, *WElement* is the base element from which all other elements inherit. It has a name and a description. *WModel* represents the root element that contains all model elements. *WLink* denotes the link. *WLink* has a

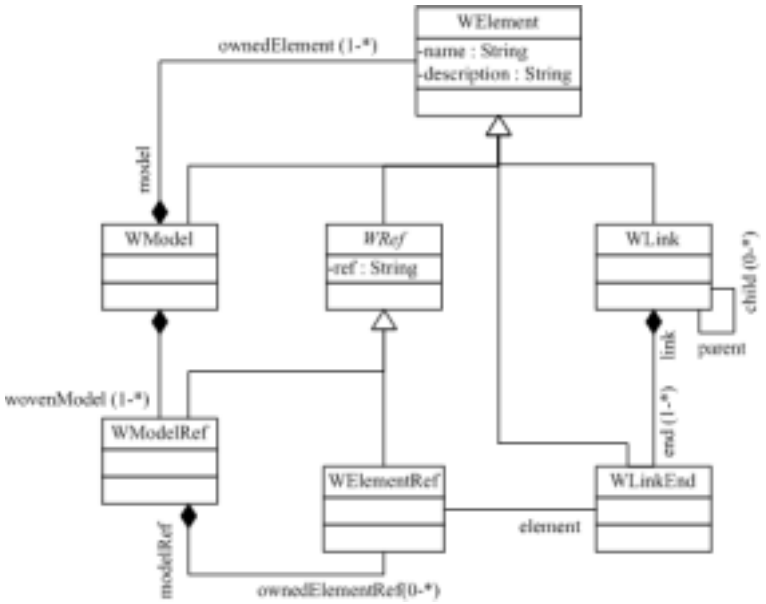

Figure 2: The core weaving metamodel

reference end to associate it with a set of link endpoints (*WLinkEnd*). The cardinality of the reference shows it supports linking 0:N elements, i.e, more than only 1-to-1 relationships. *WLink* can also have children links. For instance, a link between two attributes can be child of a link between two classes, creating nested links. Every *WLinkEnd* references one *WElementRef*. The attribute *ref* contains a unique identifier of the linked elements. It is possible to refer to the same model element by different link endpoints, e.g., one model element may participate in more than one linking expression. *WModelRef* is similar to *WElementRef*, but it contains references to the models as a whole.

This metamodel has been designed to be generic. All the classes in it are abstract, and thus, these classes need to be specialized to tailor the metamodel to the needs of transformation specification. This metamodel can also be refined to various scenarios. For instance, we can define links adapted to (the cardinalities are between parenthesis):

- Transformation specification : Rule (1:N), Binding (1:1);
- Data integration: Concatenation (1:N), Equality(1:1), IntToStr (1:1);
- Traceability: OriginSource (N:N), Modified (1:N), Added (0:1);
- Composition: Override (1:1), Merge (N:1), Delete (1:0).

We define an extension specialized to the abstract specification of transformations, as listed below. It is written using KM3 [17], which is a textual

language for defining metamodels. KM3 has similar semantics to the Ecore metametamodel [10]. Ecore is the standard format of the Eclipse Modeling Framework (EMF) [10] to define metamodels.

```
package mmw_transformation {
  class Module extends WModel{
    reference inputModels [1-*] container : InModelRef;
    reference outputModels [1-*] container : OutModelRef;
    reference rules [1-*] container : Rule;
  }
  class Rule extends WLink {
    reference input container : InputElement;
    reference output [1-*] container : OutputElement;
  }
  abstract class TModelRef extends WModelRef { }
  class InModelRef extends TModelRef { }
  class OutModelRef extends TModelRef { }
  class InputElement extends WLinkEnd {
    attribute varName : String;
  }
  class OutputElement extends WLinkEnd {
    attribute varName : String;
  }
  class Binding extends WLink {
    reference target container : ReferredElement;
    reference source container : ReferredElement;
  }
  class ReferredElement extends WLinkEnd { }
  class TransformationElementRef extends WElementRef { }
}
```

Module is the root element that stores references to the input and output models, i.e., they are links to the models as a whole. A module supports one or more input/output models. A module contains one or more rules. The *Rule* element is a link, which connects one input element to one or more output elements. The information about the link endpoints are stored in the *InputElement* and *OutputElement* elements. For instance, a *Rule* can denote a link between two endpoints (e.g., $UMLClass \leftrightarrow JavaClass$). The *varName* attribute enables assigning a symbolic name to the linked elements, used on a latter stage on the generation of transformations. The output elements can have one or more *Binding*. A *Binding* denotes a 1:1 link between *attributes* or *references* and *ReferredElement* is the endpoint of each link. This way we can have nested links between *Rule* and *Binding*s. For instance, the link $UMLClass/name \leftrightarrow JavaClass/name$ is child of the link $UMLClass \leftrightarrow JavaClass$). *ReferredElement* has a reference to the *TransformationElementRef*. It contains the *ref* attribute, which stores the real addresses (pointers) of the elements.

5.1.2. Initialization of the weaving model in AMW

AMW is a set of plug-ins built on top of the Eclipse platform [6]. It extensively uses the EMF (Eclipse Modeling Framework) API [10] for models manipulation, i.e., for creating, editing and serializing the (weaving) models and metamodels. It is also dependent of the ATL plug-ins, because the transformation skeletons are generated in the ATL language. In addition, the higher order transformations are as well written using ATL.

The main design decision behind the implementation is to have a simple user interface for AMW, based on the core weaving metamodel already presented but independent of the AMW extensions used. The interface is dynamically generated.

The interface is designed as a three-panel editor, as illustrated in Figure 3. The left panel shows the input metamodel. The right panel shows the output metamodel (the Java metamodel) and the middle panel shows the relationships between the model elements. These relationships are stored as a weaving model.

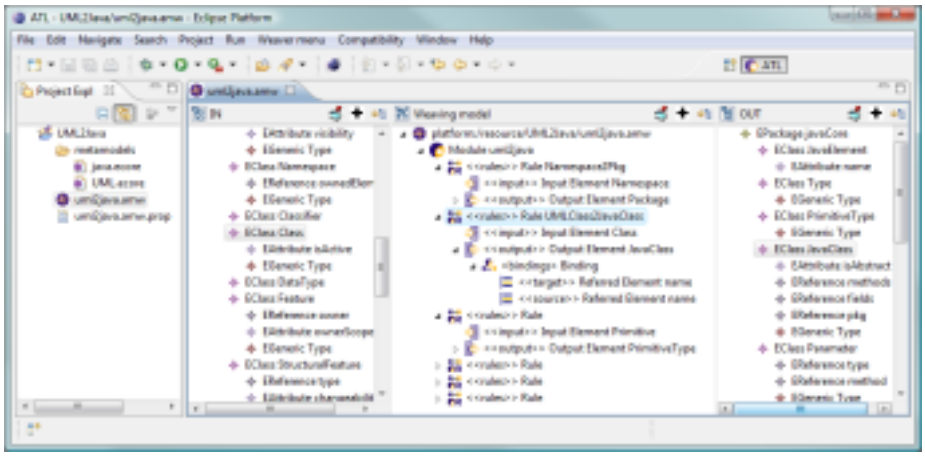

Figure 3: The three panels of AMW

The weaving models need to be configured the first time they are created. The first step is to choose the metamodel extensions, the models that are going to be linked and the panels that are used. The panels are responsible for loading the linked models. The default panel is a tree-based panel, but graphical interfaces could be chosen as well, if available. These configurations are done with the help of a three-page wizard. This wizard can be called in the "File-New-Model Weaver-Weaving Model" menu. The weaving model can be created in any Eclipse perspective, however, it is recommended that it is loaded in the "ATL Perspective" specially because the models need to have been created as part of an *ATL Project* to facilitate the generation of the transformation skeleton later on.

In the first page (see Figure 4), the developer chooses the metamodel extension. The file containing the appropriate extension for our scenario is the

mmwqatl.km3, which is an implementation of the extension presented in section 5.1.1.

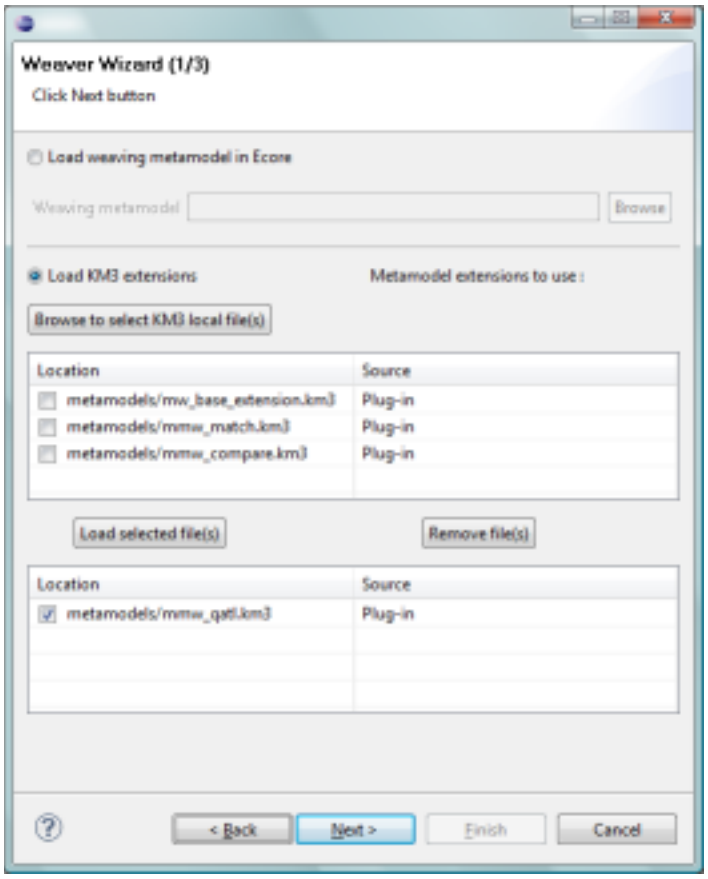

Figure 4: Wizard first page

The wizard second page (see Figure 5) has three main parts. First, we specify the containing project and the name of the weaving model, with the extension *.amw*.

Second, we choose the implementation of the panel that manipulates the weaving model (called weaving panel). The standard implementation is a tree-based panel, called *DefaultWeavingPanelExtension*. This weaving panel has menus for creating and modifying the weaving elements, such as the property editor, or the creation of links, compositions and references.

We need to choose the *TransformationWeavingPanel* implementation, because it contains functionalities targeted to the transformation specification scenario, in addition to the model manipulation adaptive menus.

This panel can be easily replaced, because AMW publishes an extension

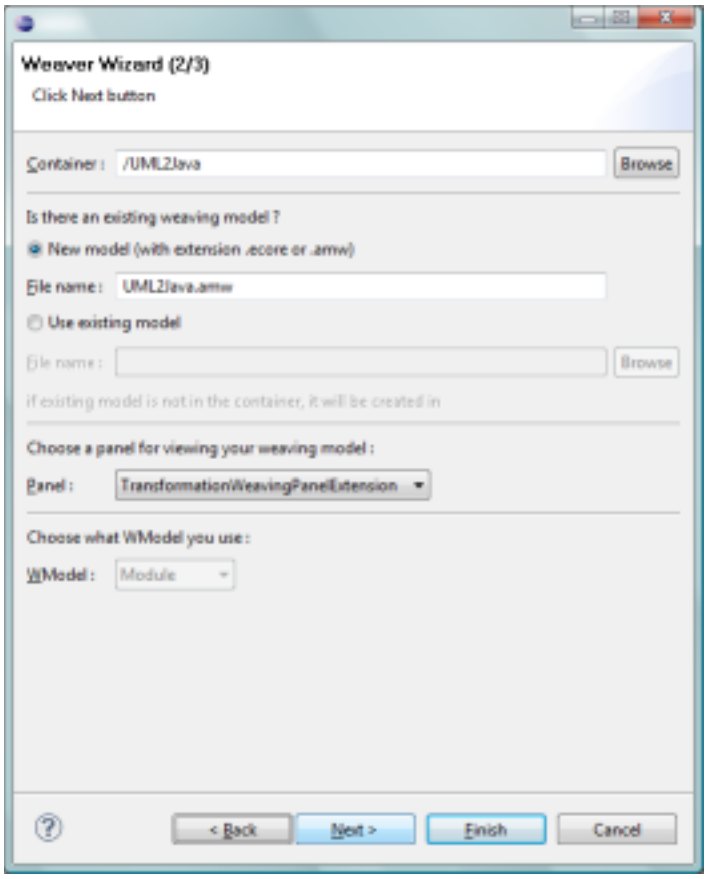

Figure 5: Wizard second page

point that enables plugging different panels. An extension point is the Eclipse mechanism that defines how plug-ins can be plugged into other plug-ins. A plug-in which defines an "extension" contributes (can be plugged) to the defined "extension point". The definition of the extension (as shown below) uses the *org.eclipse.weaver.weavingPanelID* extension point, and the plugged panel is implemented by the class *org.eclipse.gmt.weaver.transformation.panel.TransformationWeavingPanel*.

```
<extension point="org.eclipse.gmt.weaver.weavingPanelID"
   id="TransformationWeavingPanelExtension">
   <weavingPanel
      name="Transformation weaving panel extension"
      class="org.eclipse.gmt.weaver.transformation.
            panel.TransformationWeavingPanel"/>
</extension>
```

Third, we choose the root element that is created (an extension of *WModel*). In our particular application, the AMW extension has only the *Module* element that extends WModel but in general we could have several ones.

The third wizard page (see Figure 6) configures the to-be linked (woven) metamodels. Similarly to the weaving model, AMW provides a standard panel implementation for manipulating the linked metamodels (or models). The panels are plugged by using to the *org.eclipse.weaver.wovenPanelID* extension point. The standard implementation is the class named *org.eclipse.weaver.extension.panel.DefaultWovenModelPanel*. AMW enables linking as many models as defined in the *Module* element. The UML2Java scenario has only one input and one output metamodel, however, we could have more than one output metamodel, as defined in the cardinality of the *outputModels* reference. The names of the metamodels are shown on the top of the corresponding panels.

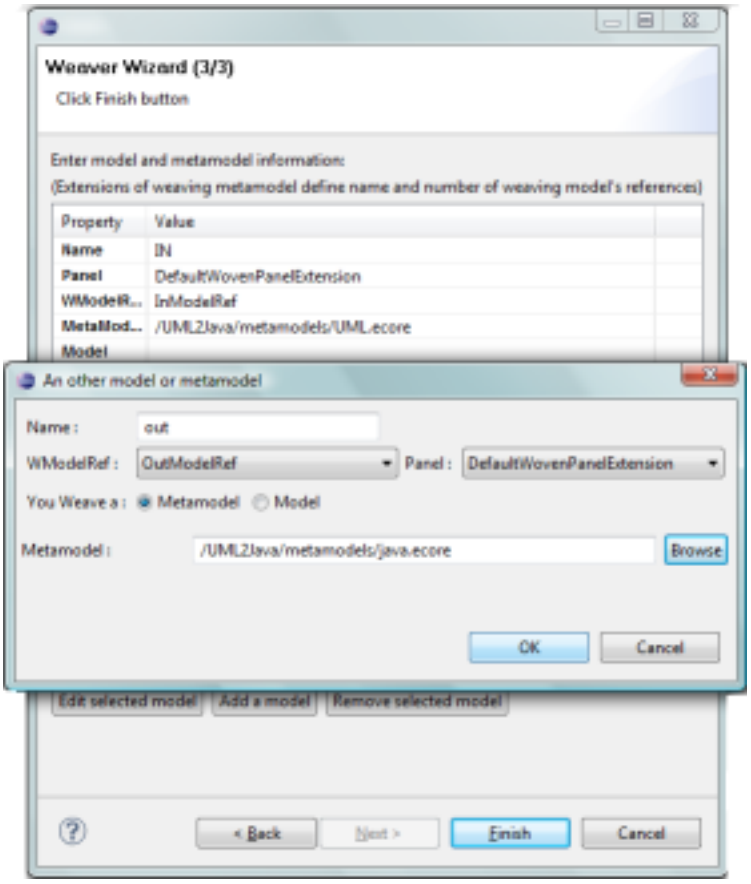

Figure 6: Wizard third page

For each metamodel (or model), the pop-up window enables defining the

WModelRef extensions that are used and the path of the metamodels (or models). The elements that extend *WElementRef* are associated with the *dereferencing classes*. These classes read/write the value of the *ref* property and return the corresponding model element. This way, AMW can link elements with different identification mechanisms. For instance, the default provided implementation uses the *XPointer* value produced by the EMF API. However, it is also possible to use the *XMI:ID*.

The dereferencing classes must implement the *IIdentifierAdapter* interface shown below. The *setId(Object obj)* method assigns a unique identifier to a given model element. The *getId()* method returns the identifier of a given element.

```
public interface IIdentifierAdapter extends Adapter {
   public void setID(Object obj);
   public Object getID();
}
```

The association between the dereferrncing classes and the model elements are set up using an extension point (an example is shown below). The attribute *adapterClassName* contains the name of the type of the model element. This means that every time the *setID* and *getID* operations are called over an element with type *ElementRef*, it calls the wrapped methods from the *ElementRefItemProvider* class.

```
<extension point="org.eclipse.weaver.itemProviderID"
  id="ItemProviderExtension">
  <itemProviderAdapter
    name="Base Item provider extension"
    class="org.eclipse.weaver.extension.providers.
     ElementRefItemProvider"
    adaptedClassName="TransformationElementRef"/>
</extension>
```

The wizard is only executed when the weaving model is created for the first time. To open an already configured model, the developer can just double-click on the appropriate AMW file. The interface will show up with the right panels, metamodels and weaving model as indicated. AMW is ready to edit the transformation specification.

5.1.3. Editing the weaving model

In this section, we describe how to edit the weaving models previously configured. There are two main mechanisms to edit a weaving model: using the dynamically generated menus or drag-and-drop of elements.

The weaving model initially contains only the root element *Module*. We use the pop-up contextual menus available for every element to create other elements, as shown in Figure 7. First, we add a *Rule* using the menu. For each *Rule*, we drag and drop the elements from the left and right metamodels over the corresponding Rule link. AMW suggests which link endpoints can be

created. In our scenario, we create one *InputElement* for the UML class and one *OutputElement* for the Java Class. For every output element, we create bindings between the UML structural features (attributes or references) and the corresponding Java fields. Other rules can be created in the same way. In addition, the rules can also be created by copying and pasting other rules. In this case, we need only to redefine the linked elements accordingly.

The menus are dynamically generated based on the elements of the weaving metamodel extension. For instance, AMW creates menu items for all elements that inherit from *WModel*, *WLink* and *WLinkEnd*. Consequently, it provides menus for creating the *Module*, *Rule*, *Binding* and *ReferredElement*. Since the implementation is based on the reflective API of EMF, more kinds of links can be added. This is very important to obtain the appropriated level of abstraction. For instance, we could add a new kind of link that represents the concatenation between model elements, and call it *Concatenation*. AMW will generate the menus accordingly.

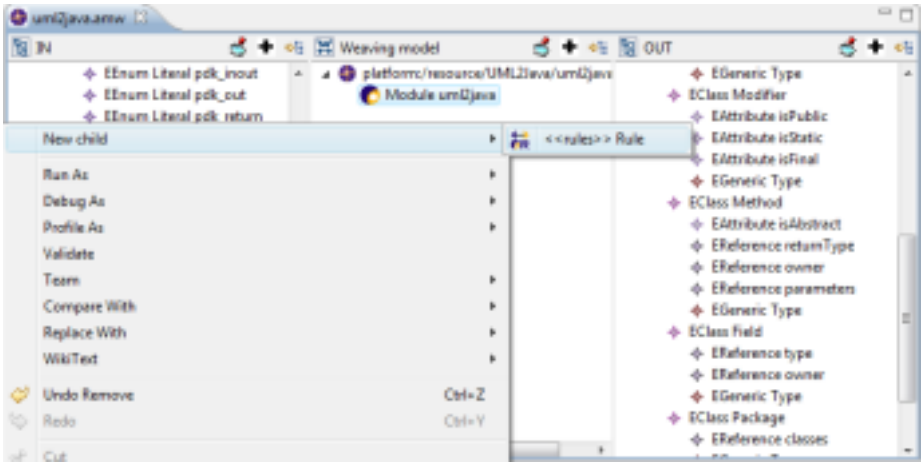

Figure 7: Menus for creating elements

After the rules are created, we can browse the weaving model by 1)expanding/collapsing rules and bindings and 2) clicking on the links. When we click on the links, the linked elements from the left and right panels are highlighted. Once the specification is completed, the weaving model is used to produce the transformation skeleton in ATL.

5.2. Producing the transformation skeleton : level 2

In this section, we present how we implement the operation of derivation of weaving models into transformation skeletons. First, we present the concept of Higher-Order Transformations, which are used to implement the derivation operation. Them, we show how they are plugged and executed by AMW.

5.2.1. Higher-Order Transformations (HOTs)

The operations that transform one abstract representation (a weaving model) into a more concrete one (a transformation skeleton) are implemented using Higher-Order Transformations (HOT). A higher-order transformation is a model transformation, such that the input and/or the output models are transformation models [7].

We show below an excerpt of an HOT in ATL (the full HOT has 278 lines). In the scenario of transformation specification, a HOT is a model transformation that takes the weaving model (IN), the *left* and *right* metamodels as input and that produces the ATL skeleton (OUT) as output. The left and right metamodels conform to the *Ecore* metametamodel (we use the *MOF* word to keep compatibility with ATL). The output skeleton conforms to the ATL metamodel [15].

```
module WeavingSpecToATL;
create OUT : ATL from IN : AMW, left: MOF, right: MOF;
rule TransformRule {
  from
    amw: AMW!Rule
  to
    atl : ATL!MatchedRule (
      name <- amw.name,
      isAbstract <- amw.isAbstract,
      inPattern <- amw.input,
      outPattern <- amw.output
}
rule InPatternElement {
  from -- source element
    amw: AMW!InputElement
  to
    atl: ATL!InPattern (
      elements <- element
    ),
    element : ATL!SimpleInPatternElement(
      varName <- amw.getVarName('in_'),
      type <- amw.left.element.ref
    )
}
rule OutPatternElement {
  from -- target element
    amw: AMW!OutputElement
  to
    outelement : ATL!SimpleOutPatternElement(
      varName <- amw.getVarName('out_'),
      type <- amw.left.element.ref
  )
```

```
}
```

This HOT has three rules. The first rule transforms every *Rule* element of a weaving model into an ATL *MatchedRule*. It assigns a value to the *isAbstract* attribute. It also implicitly assigns the references (input and output patterns). These elements are transformed by the second and third rules. The second rule matches an AMW *InputElement* and produces an ATL input pattern. The second rule matches an AMW *OutputElement* and produces an ATL output pattern. For instance, if we have one relationship between two elements, this rule produces one ATL rule, with the *(to - InPattern)* and *(from - SimpleOutPatternElement)* parts.

5.2.2. Plugging and executing the HOT's in AMW

AMW provides an extension point for plugging and executing more than one HOT. This brings the possibility of writing different HOTs for generating different output transformations. This is particularly important for generating more than one skeleton and when new kinds of links are added into the weaving metamodel.

The extension point configuration (i.e., the parameters defining where the transformation is plugged) is illustrated below. The *id* attribute has an unique identifier. The *point* attribute contains the name of the extension point. The *hot* element configures the HOT transformation. It contains the transformation path and a menu group and a label that will be shown in th user interface. The bindings define the correspondences between the transformation signature (*left, right, MOF*) and the references from the weaving metamodel (*inputModels, outputModels* from the *Module* element).

```
<extension id="Weaving2ATL"
   point="org.eclipse.gmt.weaver.transformation.transformationID">
   <hot transformation="transformations/WeavingSpecToATL.asm"
      description="Weaving specification to ATL" file_suffix="atl"
      category="AMW to ATL" >
      <binding weavingReference="inputModels" header="left"
         metamodelHeader="MOF"/>
      <binding weavingReference="outputModels" header="right"
         metamodelHeader="MOF"/>
   </hot>
</extension>
```

The interface generates an entry in the pop-up menu for executing the HOT, as shown in Figure 8. When the user clicks on it, the HOT generates and stores an ATL transformation, in the same folder of the weaving model. The transformation is stored using its textual format (i.e., an .atl file). We show below two ATL rules that are generated. The full transformation definition is available in the AMW web site.

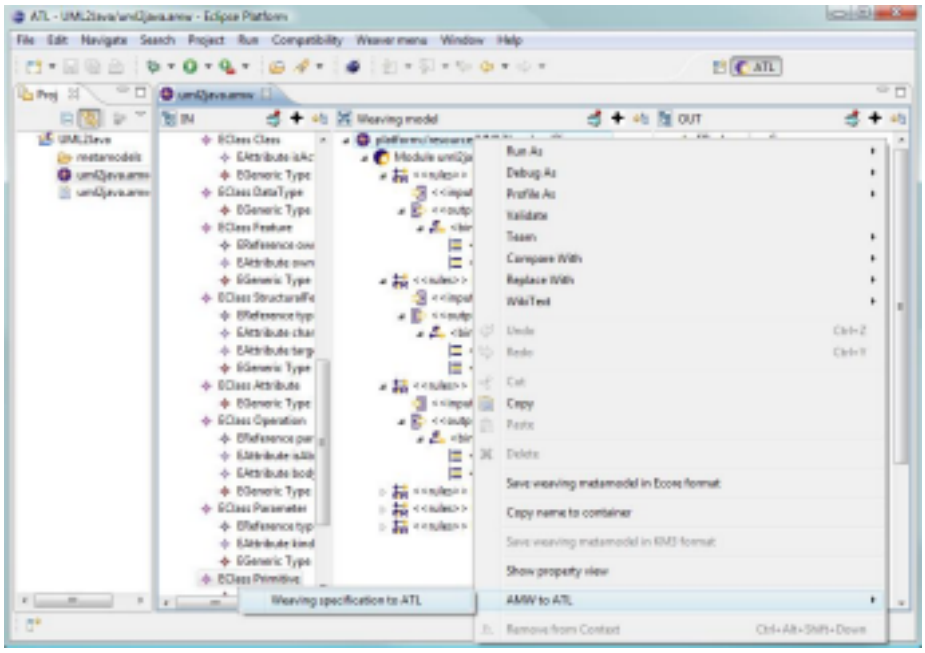

Figure 8: Generation of the transformation skeleton

```
module uml2java;
create OUT_model : OUT from IN_model : IN;

rule Namespace2Pkg {
 from
  in_namespace : IN!Namespace
 to
  out_package : OUT!Package (
   name <- in_namespace.name
  )
}
rule UMLClass2JavaClass {
 from
  in_class : IN!Class
 to
  out_javaclass : OUT!JavaClass (
   name <- in_class.name
  )
}
rule Primitive_2_PrimitiveType {
 from
  in_primitive : IN!Primitive
```

```
 to
  out_primitivetype : OUT!PrimitiveType (
    name <- in_primitive.name
  )
}
```

Once the skeleton is generated, the last step (*level 1*) is the refinement of this skeleton for producing the final ATL code [5]. After that, the transformation can be executed in the ATL engine to transform UML classes into Java elements.

6. Extending AMW with link discovery techniques

The manual creation of the weaving models may become a tedious task when the metamodels have similar structure and/or are big. A large part of the relationships are often simple links between model elements with similar semantics. This can be seen on links such as $Class \leftrightarrow JavaClass$, $Class/name \leftrightarrow JavaClass/name$ or $DataType \leftrightarrow Type$.

In this section, we present how we extend AMW to semi-automate the creation of links between model elements. The process of semi-automating the creation of relationships is called *matching* [18]. The matching process consists of combining different methods that estimate a similarity value between the model elements and determine that the elements constitute a match when this value goes over a given threshold. A simple matching process is typically composed by three steps:

1. Creation of links between all pair of elements from the input and output metamodels (this can be seen as a Cartesian Product).
2. Assignment of a similarity value between elements, e.g. focusing on those that have similar names and using a String-distance method to calculate the similarity.
3. Filtering and selection of the links that have a similarity higher than a given threshold.

There is extensive work about matching of database/XML schemas or ontologies (see [18] for a detailed survey). We take advantage of these approaches when extending AMW with a semiautomatic matching support.

The extension of AMW with semiautomatic matching involves two main tasks: 1 - definition of a new weaving metamodel extension that enables the storage of a similarity value between two linked elements and 2 - the implementation of different predefined matching methods for creating and refining the weaving models conforming to this extension.

[5] We don't show how to edit the transformation in ATL because the ATL IDE is an independent plug-in, with its own interface, documentation and user community: http://www.eclipse.org/atl

6.1. Metamodel extension for automatic link discovery

We define a weaving metamodel extension that enables the establishment of links with a similarity value between classes, attributes and references (as illustrated below).

```
package mmw_match {
   class MatchModel extends WModel {
      reference leftM container : WModelRef;
      reference rightM container : WModelRef;
   }
   abstract class Equal extends WLink {
      attribute similarity : Double;
      reference left container : WLinkEnd;
      reference right container : WLinkEnd;
   }
   class ElementEqual extends Equal {}
   class AttributeEqual extends Equal {}
   class ReferenceEqual extends Equal {}
   class LeftElement extends WLinkEnd {}
   class RightElement extends WLinkEnd {}
   datatype Double;
}
```

MatchModel is the root. It contains two metamodels (*leftM* and *rightM*). The three extensions to the *Equal* links store relationships between two elements with the same type. The *similarity* attribute stores a float value that is an estimation of the proximity. The links can contain child links (using the *child* reference from the *WLink* element), for instance, a *ReferenceEqual* link can be contained by a *ElementEqual* link.

We do not reuse the metamodel described in section 5.1.1 due to two reasons. First, the new metamodel has a similarity value that is only applicable to the semi-automatic process. Second, using different link types - independent of the transformation language - shows that the interface can be dynamically adapted.

To use this metamodel extension, in the first wizard page, it is necessary to choose the following three files: *mmwmatch.km3*, *mwbaseextension.km3* and *mmwcompare.km3*: they contain links used in the automatic discovery process. In the second weaving page, the settings are the same as for the manual creation. In the third wizard page, the linked models are pre-configured into *leftM* and *rightM*. It is not necessary to add new model, but just to edit the proposed ones (the proposition is based on the cardinality of the references).

6.2. Configuration of the matching process

The matching process is separated in several steps, which are sequentially executed. Each step is implemented by a specific model transformation, called *matching transformation* [19]. A matching transformation takes a weaving

model, a left and a right metamodel as input and produces a new weaving model as output.

Each matching transformation refines the previous similarity value (if existing). Consider for instance three matching transformations T1, T2 and T3. T1 produces a weaving model; T2 takes the result of T1 and produces a new weaving model with new similarity values; T3 takes the result of T2 and produces a new weaving model as well. All these weaving models conforms to the weaving metamodel with the *mmwmatch* extensions.

AMW provides an extension point where different matching transformations can be plugged. The configuration screen (as shown in Figure 9) enables combining different transformations for the matching process and sets up their execution parameters.

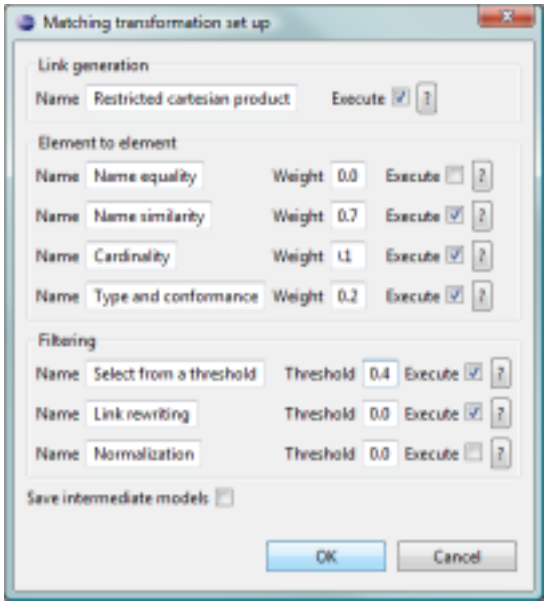

Figure 9: Configuration of the matching transformations

The configuration window has one group for each different kind of matching transformation. Each group shows the set of available parameters. The parameters are:

- *execute*: selects the execution of a transformation
- *weight*: assigns the influence of the matching transformations on the overall similarity estimation;
- *threshold*: selects only the links with the estimation higher than its value.

The "? (question mark)" button shows the required metamodels for executing the transformation. The "Save intermediate models" button saves a new weaving model after the execution of each matching transformation. This enables the comparison of the intermediate results.

The configuration shown in Figure 9 is an example setting we use for a typical matching process. It executes the following transformations: a restricted Cartesian product (i.e., it matches only elements with the same type); a comparison over the elements names, with a weight of 0.8; a comparison over the elements cardinality, with weight 0.1; a comparison of the type, with weight 0.2; a selection of the links with similarity value higher than 0.4; and the rewriting of links with better similarity values. The weights are set up empirically, i.e., we assume that elements with the same name have a higher probability of being similar than elements that have the same cardinality or type.

The transformations are executed following the sequence indicated. The result is a weaving model with a set of links. Generally, this set of links is smaller than the initial one, because we discard links with low similarity values. In order to improve the accuracy of this process, the value of the weights can be changed and new matching transformations can be plugged. The choice of the right weight values and transformations is a trial-and-error process, based on several executions.

This weaving model can be manually refined, by adding new links or deleting existing ones. Finally, it is used to produce an ATL skeleton as well, following the same process described on section 5.2.2.

7. Related Work

Several research works have addressed the simplification of the specification of model transformations. However, they are not based on an explicit separation of the transformation development into several steps at different abstraction levels and most of them do not provide adequate tool support (specially with respect to tools that can be easily installed and integrated within the Eclipse platform).

QVT relations [2] are bidirectional mappings between model elements. A QVT relation may also have a guard to restrict the elements that are mapped. QVT relations are part of the QVT specification as a high-level definition of operational mappings, which are the executable transformations. It differs from our approach because the metamodel is fixed and the language is purely textual. There are different tools implementing QVT-R, such as Declarative QVT [20], MediniQvt [21], however, these tools are on early development stages.

UML-QVT [22] and Operational QVT [23] are the operational specifications of the QVT standard. However, the transformation from QVT Relations to QVT Operational Mappings remains unsolved [24]. To the best of our knowledge, there is not a QVT-like tool that supports the complete transformation specification process.

The solution from [25] presents a graphical transformation language. Instead of explicitly separating all the three specification levels, the graphical

constructions are combined with textual expressions. For that reason, the specification is complete. This means the final transformation (level 1) is generated directly from these specifications, without having a skeleton. The authors have also developed ModeLink [26], which is used to establish relationships between model elements. The principle is similar to the three-panel interface of AMW. However, this tool is targeted essentially to traceability scenarios.

The work from [27] presents a solution to help on the creation of model transformations separated in three levels. However, the relationships are between terminal models, not metamodels. MTBE (Model Transformation By Example)[28] also uses relationships between terminal models. The model transformation is derived from these relationships. These approaches can be effective when using terminal models with reasonable size. They must have at least one sample of the input and output models. In [29], the authors derive the approach of MTBE, but the transformations are specified using two levels (complete specification and transformation into executable format). The tool is not available for comparison.

Triple Graph Grammars (TGGs) [30] are a solution for defining correspondences between two models, similar to our weaving models. Fujaba [31] is a tool that provides an implementation of TGGs. The TGGs act as specifications to the Fujaba's executable format. The TGGs have a concrete graphical syntax, where the correspondences are established manually.

8. Conclusions

In this paper we have presented AMW, a tool for the multi-level specification of model transformations and shown its usefulness when dealing with complex transformation scenarios, using the UML2Java transformation as illustrative example.

AMW is composed by a set of plug-ins on top of the Eclipse framework. The integration in Eclipse enables using it in parallel with a rich set of modeling plug-ins. The AMW site [6] provides extensive documentation, the source code, a set of predefined metamodel extensions and a set use cases to quickly install and test the tool.

As future work, we plan to study new extensions of the weaving metamodel. In particular, we would like to provide "packaged extensions" of the metamodel that include predefined sets of relationship types for specific application scenarios (e.g. refines, complements,...), to use AMW in a forward-engineering process management scenario as requested by interested industrial partners. We also plan to empirically validate the limits of the graphical interface specification of the mappings depending on the kinds of links and model size. Finally, we plan to use the AMW extension points to plug several algorithms for link discovery that help in the semi-automatic creation of weaving models.

[6] http://www.inf.ufpr.br/didonet/amw/

References

[1] F. Jouault, I. K. I, Transforming Models with ATL, in: MoDELS Satellite Events, 2005, pp. 128–138.

[2] Object Management Group, Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT) Specification, version 1.0 (2008).

[3] K. Ehrig, E. Guerra, J., L. Lengyel, T. Levendovszky, U. Prange, G. Taentzer, D. Varró, S. Varró-Gyapay, Model transformation by graph transformation: A comparative study, in: MTiP 2005, International Workshop on Model Transformations in Practice (Satellite Event of MoDELS 2005), 2005.

[4] Kermeta language. Reference Manual. http://www.kermeta.org/docs/KerMeta-Manual.pdf (2009).

[5] R. S. Pressman, Software Engineering: A Practitioner's Approach, 5th Edition, McGraw-Hill Higher Education, 2001.

[6] Eclipse Foundation : http://www.eclipse.org (2010).

[7] M. D. D. Fabro, Metadata management using model weaving and model transformations, Ph.D. thesis, University of Nantes (2007).

[8] OMG: Object Managment Group (2010).

[9] UML 2.1.2 specification: http://www.omg.org/spec/UML/2.1.2/ (2007).

[10] EMF : http://www.eclipse.org/modeling/emf/ (2010).

[11] MOF 2.0 Specification: http://www.omg.org/spec/MOF/2.0/ (2006).

[12] K. Czarnecki, S. Helsen, Feature-based survey of model transformation approaches, IBM Syst. J. 45 (2006) 621–645.

[13] M. Tisi, F. Jouault, P. Fraternali, S. Ceri, J. Bézivin, On the use of higher-order model transformations, in: ECMDA-FA, 2009, pp. 18–33.

[14] Oracle, The Java Pet Store reference application. http://java.sun.com/developer/releases/petstore/ (2010).

[15] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, Atl: A model transformation tool, Sci. Comput. Program. 72 (1-2) (2008) 31–39.

[16] ATL Transformation repository : http://www.eclipse.org/m2m/atl/ atl-Transformations/ (2010).

[17] F. Jouault, J. Bézivin, Km3: A dsl for metamodel specification, in: FMOODS, 2006, pp. 171–185.

[18] P. Shvaiko, J. Euzenat, A survey of schema-based matching approaches (2005) 146–171.

[19] M. D. D. Fabro, P. Valduriez, Towards the efficient development of model transformations using model weaving and matching transformations, Software and System Modeling 8 (3) (2009) 305–324.

[20] Eclipse, Declarative QVT. http://www.eclipse.org/m2m/ (2011).

[21] ikv++, Medini QVT. http://projects.ikv.de/qvt/ (2011).

[22] UML-QVT. http://umt-qvt.sourceforge.net/ (2011).

[23] Borland, Operational QVT, http://www.eclipse.org/m2m/qvto/doc/ (2011).

[24] R. Romeikat, S. Roser, P. Müllender, B. Bauer, Translation of qvt relations into qvt operational mappings, in: ICMT, 2008, pp. 137–151.

[25] E. Guerra, J. Lara, D. Kolovos, R. F. Paige, A visual specification language for model-to-model transformations, in: In VL/HCC, 2010, pp. 119–126.

[26] ModeLink : http://www.eclipse.org/gmt/epsilon/doc/modelink/ (2010).

[27] D. Varró, Model transformation by example, in: MoDELS, 2006, pp. 410–424.

[28] M. Wimmer, M. Strommer, H. Kargl, G. Kramler, Towards model transformation generation by-example, in: Proceedings of the 40th HICSS, IEEE Computer Society, Washington, DC, USA, 2007, pp. 285b–.

[29] Y. Sun, Supporting model evolution through demonstration-based model transformation, in: Proceeding of the 24th OOPSLA, OOPSLA '09, ACM, New York, NY, USA, 2009, pp. 779–780.

[30] A. Schürr, F. Klar, 15 years of triple graph grammars, in: ICGT, 2008, pp. 411–425.

[31] Fujaba Tool Suite. http://www.fujaba.de/ (2011).