Refining Models with Rule-based Model Transformations

Massimo Tisi, Salvador Martínez, Frédéric Jouault, Jordi Cabot

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Refining Models with Rule-based Model Transformations

Massimo Tisi — Salvador Martínez — Frédéric Jouault — Jordi Cabot

N° 7582

Mars 2011

Thème COM

ISSN 0249-6399 ISRN INRIA/RR--7582--FR+ENG

Refining Models with Rule-based Model Transformations

Massimo Tisi , Salvador Martínez , Frédéric Jouault , Jordi Cabot

Thème COM — Systèmes communicants
Projets AtlanMod

Rapport de recherche n° 7582 — Mars 2011 — 18 pages

Abstract: Several model-to-model transformation languages have been primarily designed to easily address the syntactic and semantic translation of read-only input models towards write-only output models. While this approach has been proven successful in many practical cases, it is not directly applicable to transformations that need to modify their source models, like refactorings. In this paper we investigate the application of a model-to-model transformation language to in-place transformations, by providing a systematic view of the problem, comparing alternative solutions and proposing a transformation semantics to address this problem in ATL.

Key-words: transformation, refining, model

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 Rennes Cedex (France)
Téléphone : +33 2 99 84 71 00 — Télécopie : +33 2 99 84 71 71

Raffinage de modèles à l'aide de transformation de modèle basé sur les règles

Résumé : Une partie des langages de transformation de modèle a été définis pour réaliser facilement la traduction syntaxique et sémantique de modèles d'entrée en lecture seule vers des modèles de sortie en écriture seule. Bien que cette approche ai fait ses preuves dans de nombreux cas pratiques, elle n'est pas directement applicable pour les transformations devant modifier leurs modèles d'entrée, comme pour les opérations de refactoring. Dans cet article, on étudie l'adaptation d'un langage de transformation de modèle au problème posés par les transformation en place, en fournissant une approche systématique du problème, en comparant les solutions alternatives et en proposant une sémantique de transformation permettant la résolution de ce problème en ATL.

Mots-clés : transformation, raffinage, modèle

1 Introduction

Most application scenarios of Model-Driven Engineering require at some point to automatically generate new models from existing ones, a process called model transformation. For a subset of these transformations the output model is mostly identical to the input model, with the exception of a set of changes that is relatively small compared to the model size. In this paper we refer to this special kind of transformations by the term *refinement transformations* (or simply *refinements*). An example of such refinement transformations would be any implementation of model refactorings[4, 16, 10].

Since models can be considered a particular kind of graphs, a natural way to address model refinement in a rule-based way, is reusing existing graph transformation languages. In such languages the transformation is structured in the recursive application of graph-rewriting rules on the input model. Indeed several approaches make use of graph transformations for model refinements ([19, 7, 1]). While these methods are very powerful and generic, they may look too complicated for practical refinement tasks for a non-expert user. They are based on formal frameworks, and the user needs to have at least a superficial knowledge of the formal background to avoid to incur in theoretical problems like rule conflicts, termination and confluence.

Moreover, graph transformations are natively recursive, meaning that rules are executed over the processed model until a termination state is reached. While recursive rule application can be the natural solution for several problems, several others require a more controlled application, e.g. one-step, of the transformation. Applying graph transformation languages to these problems requires the developer to explicitely disable recursion using technical means (e.g. ad-hoc negative application conditions or the introduction of supporting nodes). These workarounds are not only tedious and error-prone, but they hamper the abstractness in representing the transformation logic. Also, integration of graph transformation tools in popular modeling frameworks like EMF is still limited.

In this paper we propose an alternative solution for rule-based refinement transformations. This solution integrates some of the benefits of graph-transformation approaches on already existing popular Model-to-Model Transformation Languages (M2MTL) to provide designers with an easy and efficient approach for the definition of refinement transformations.

M2MTLs are domain specific languages born to allow the translation, in a simple and abstract way, of one (or more) input models into one (or more) output models. The translation is performed at the same level of abstraction of the manipulated models, by talking about model elements as first-class citizens. In this way the transformation does not require the encoding of models in specific formalisms like graphs, XML trees or textual serializations. The transformation logic results more abstract, being clean from technical details about the particular representation of the model.

To keep their semantics relatively simple the most popular M2MTLs like the AtlanMod Transformation Language (ATL)[3] or OMG Query View Transformation (QVT)[14] make strong assumptions on the direction of model parameters, forbidding a model to be both input and output of the transformation. In this way, rules are never applied recursively to their own output. This design choice makes M2MTLs a more natural language to address non-recursive problems, freeing the developer from caring about disabling recursivity when not needed. Moreover this simplification has the benefit of making transformations immune to the rule-conflicts and termination problems of graph transformations. Real-world experience has shown that this simplicity, that appears to reduce

the computational power of M2MTLs, does not hamper their practical applicability to common usage scenarios. This is, for example, pointed out by the number of applications in the Eclipse.org ATL transformation Zoo[6].

Nevertheless, the application of these M2MTLs to refinement transformations is not straightforward: refinement transformations do not strictly adhere to the input-or-output assumption of the model-to-model paradigm because they need to modify their own source model. If developers want to use current M2MTLs languages, they can only simulate a model refinement by writing a transformation that 1) copies all the source model elements into a new target model, and 2) makes the small changes while performing the copy. However, in this paper we will show that this naive solution has important drawbacks: 1) The development of the copy rules is tedious, the transformation code unnecessarily bulky, and the change logic is drowned in the copying code, making the refinement transformation difficult to read; 2) The execution time of the refinement increaese, since matching and copying all the unchanged elements is done anyway; and 3) This semantics is incompatible with some important use cases, like the modification of a model for live refactoring. Live refactoring requires to keep the element identifiers unchanged to maintain the live links with the environment. This is only possible by directly operating on the input model, without performing a copy of it.

For these reasons this paper aims to show that a special support to refinement can make M2M transformations more suitable to refinement cases. In the process, we investigate the general problem of applying model-to-model transformation languages to refining transformations. We also provide a prototype implementation of the proposed approach as a novel in-place semantics for the ATL refining mode.

The paper is organized as follows. Section 2 introduces the running case of the paper. Section 3 lists the design dimensions and possible strategies for implementing a refinement support in M2MTLs. Section 4.1 illustrates the different kinds of support that ATL has provided to refinement transformations so far. Section 4.2 describes the new refinement semantics for ATL proposed in this paper. Section 5 investigates the related work and, finally, Section 6 gives some concluding remarks.

2 Motivating example

As a running example for this paper, we introduce a refining transformation over a simplified version of a Class Diagram, conforming to the metamodel in Figure 1. The *ClassDiagram* metamodel describes a minimal class diagram, including classes, attributes, operations and datatypes. Each *Class* contains ordered sets of *Attributes* and *Operations*. Attributes are instances of a Class or DataType and Operations have a specific *returnType* (we omit parameter passing in this example). The scope of Attributes and Operations is defined by the *isPrivate* boolean attribute.

As an example of refinement we consider the *Public2PrivateAndGetter* transformation. It is a small refactoring that modifies a public attribute of a given class, by making it *private* and adding to the class the corresponding *getter* operation to retrieve its value.

Public2PrivateAndGetter is clearly a simple one-step model transformation, that does not require the recursive structure of a graph transformation language. Therefore, the transformation can be implemented using a traditional M2M transformation language. That way, the initial version of the class diagram will be considered as the input model and the privatized version as the output model.

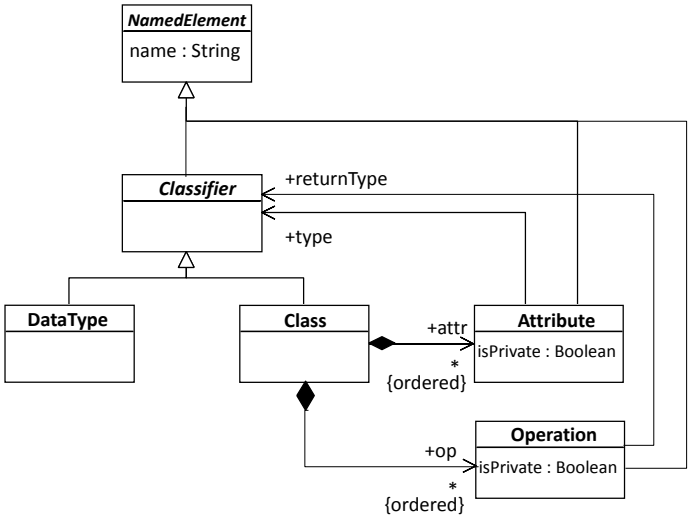

Figure 1: Source metamodel: *ClassDiagram*

Implementing a refining transformation using the standard M2M semantics requires to explicitely write the code managing the unchanged part of the input model. In a rule-based language this involves writing: 1) simple rules to copy the elements of the types not influenced by the refinement and 2) more complex rules that explicitely match and copy the unchanged elements whose types are involved in the refinement. For example listing 1 shows the standard ATL rules for the *Public2PrivateAndGetter* refactoring. In the example:

- The rule *PublicAttribute* contains all the refactoring logic, matching the public properties that don't have an associated getter and copying them to the output, changing their visibility. The same rule generates a new getter operation setting its name and return type.
- The rules *CopyClass*, *CopyOperation* and *CopyDataType* are simple rules, that copy every property of the matched element.
- The rule *CopyAttribute* is a copy rule too, but it explicitely matches the attributes not involved in the refactoring, by inverting the guard of *PublicAttribute*.

However, when using current M2M languages, designers will need to face several problems:

- As shown in the example, even in very simple cases, using the standard transformation semantics for refinement results in long code (at least a rule for each concrete metamodel class) where the refinement logic is mixed with the copying logic. The parts of the transformation that actually perform changes are obfuscated by the rules that are just copying elements. This makes the refinement logic difficult to understand.

Listing 1: Public2PrivateAndGetter.atl

```
rule CopyClass {
  from
    s : ClassDiagram!Class
  to
    t : ClassDiagram!Class (
      name <- s.name,
      attr <- s.attr,
      op <- s.op
    )
}

rule PublicAttribute {
  from
    s : ClassDiagram!Attribute (
      not s.isPrivate and
      not s.owner.op->exists(o | o.name = 'get' +
        s.name.toUpperCase() and o.returnType = s.type)
    )
  to
    t : ClassDiagram!Attribute (
      name <- s.name,
      owner <- s.owner,
      isPrivate <- true,
      type <- s.type
    ),
    getter : ClassDiagram!Operation (
      name <- 'get' + s.name.toUpperCase(),
      owner <- s.owner,
      isPrivate <- false,
      returnType <- s.type
    )
}

rule CopyAttribute {
  from
    s : ClassDiagram!Attribute (
      not (s.isPrivate and
      not s.owner.op->exists(o | o.name = 'get' +
        s.name.toUpperCase() and o.returnType = s.type))
    )
  to
    t : ClassDiagram!Attribute (
      name <- s.name,
      owner <- s.owner,
      isPrivate <- s.isPrivate,
      type <- s.type
    )
}

rule CopyOperation {
  from
    s : ClassDiagram!Operation
  to
    t : ClassDiagram!Operation (
      name <- s.name,
      owner <- s.owner,
      isPrivate <- s.isPrivate,
      returnType <- s.returnType
    )
}

rule CopyDataType {
  from
    s : ClassDiagram!DataType
  to
    t : ClassDiagram!DataType (
      name <- s.name
    )
}
```

- The transformation engine has to instantiate a completely new class diagram and perform the explicit copy of all the unchanged elements one by one, with a strong impact over the overall transformation time.
- The creation of a new model as output makes it difficult to integrate the transformation in UML model editors. Indeed, depending on the low-level representation of the model in memory, the copied elements in the newly created output model may not be a perfect clone. They could have some technical differences with respect to the input model, like different identifiers or different element ordering. The diagram editor could not be able to manage these unexpected changes, specially to maintain external references to those elements that now have changed their identifier.

These problems are independent on the M2MTL chosen for implementing the transformations. In the rest of the paper we will focus on the ATL language, but our considerations can be easily extended to similar model-driven languages like QVT/Relations or ETL[11] or, outside MDE, to transformation languages like XSLT. All these languages do not provide an explicit support for refining. While in XSLT the copy rules can be at least made compact by using wildcards in templates, QVT-Relations requires a significant development effort. Moreover, the above-mentioned performance hit affects all the implementations known to us.

3 Design strategies for refining mode

In the next sections we present two strategies to add an ad-hoc support for refinement in existing M2MTLs, in order to cope with the previous problems.

3.1 Copy strategy.

The quality of the transformation code can be significantly improved by introducing a small change in the transformation semantics for refinement mode. The need to write copy rules can be avoided by enforcing an implicit copy for all the elements that are not explicitely matched by any transformation rule. The new strategy differs from the standard one only because of the new default copy operation.

Listing 2 shows the only ATL rule needed by the *Public2PrivateAndGetter* refactoring in the copy strategy.

Listing 2: Public2PrivateAndGetter.atl

```
1  rule PublicAttribute {
2    from
3      s : ClassDiagram!Attribute (
4        not s.isPrivate and
5        not s.owner.op->exists(o | o.name = 'get' +
6          s.name.toUpperCase() and o.returnType = s.type)
7      )
8    to
9      t : ClassDiagram!Attribute (
10       name <- s.name,
11       owner <- s.owner,
12       isPrivate <- true,
```

```
13        type <- s.type
14      ),
15      getter : ClassDiagram!Operation (
16        name <- 'get' + s.name.toUpperCase(),
17        owner <- s.owner,
18        isPrivate <- false,
19        returnType <- s.type
20      )
21  }
```

The example shows that using this strategy the developer has to specify only a minimal set of rules, mirroring exactly the refinement logic.

As a minor drawback, the new default behavior of the copy semantics makes the deletion of part of the source model a delicate operation. The basic approach requires the developer to explicitely override the default copy for the elements to eliminate (e.g. by writing a transformation rule that matches the elements to delete and generates nothing). While the approach is completely sound for a generic M2M language, we will see in Section 4.1 that we decided to provide ATL with a more flexible deletion mechanism, by leveraging its specific resolution algorithm.

The default-copy behavior can be implemented in several ways:

by transformation composition: the transformation rules that implement the refinement logic can be factorized in a single transformation to be composed with a base copy transformation. The refinement rules will be used, by means of a general-purpose composition mechanism, to override the simple copy rules for the elements to modify.

by transformation synthesis: a transformation in the standard semantics can be automatically generated, for instance by a HOT [17] taking as input a transformation in the copy strategy.

by a modified transformation engine: a modified engine can directly read the refining rules in copy strategy and trigger the default copy behavior when needed.

Whatever implementation means is chosen, it is generally quite easy to adapt existing transformation languages to the copy semantics.

3.2 In-place strategy.

In this paper we propose an alternative semantics for M2M transformation languages, that is specifically designed for in-place refinement. The semantics overcomes the limitations of the default-copy approach, by using the model transformation language to perform direct manipulation of models.

While it is not necessary to change the syntax of the transformation language, this programming paradigm is remarkably different from the standard M2M transformation semantics. When using the in-place transformation mode, the user has to abandon the idea of constructing incrementally the target model starting from an empty one. Every in-place transformation rule operates in a pre-existing context, the source model, matching and modifying the current elements.

The transformation starts with an initial state constitued by the whole source model. Explicit transformation rules are executed, and every model element that is not explicitely manipulated by these operations is simply kept as it is. A transformation in in-place mode can be translated in a

sequence of basic operations over the source model, with three operation types (mimicking a CRUD operation set):

create: builds a new model element from scratch in one of the models.

update: updates a model element in-place in one of the models, by changing one of its features.

delete: deletes explicitely a model element in one of the models.

The implementation of the new semantics over a rule-based transformation language has to be carefully designed, to avoid interaction problems between rules. Since declarative rules in M2M rule-based languages may be matched in any order, an in-place strategy cannot be implemented by incrementally updating the source model of the transformation. The problem in this case is that rules modifying the same model elements would possibly conflict with each other, with undeterministic results.

To avoid rule conflicts we propose a two-step application process for in-place transformation rules. A first application step calculates the effect of these rules on the source model, without concretely changing the model. The second step applies all the modifications at once.

Several technical solutions can be chosen to implement the two-step semantics. One possibility is to mark the modifications on the source model without actually commiting them until the end of the transformation, so that expression evaluation can occur on the original source model by ignoring the modifications. Another possibility is to first compute the set of basic operations to perform, storing this set in an external model representation and then apply all the changes at once. Our prototype implementation of the in-place refining mode follows the second approach.

Figure 2 gives an overview of our strategy for in-place refining. In this figure, we consider a refining transformation for the `MMa` metamodel that produces a model `Ma'` from a model `Ma`. This transformation is represented as a dashed arrow. The two-step process is represented using dotted arrows, and it's divided in:

- **Change computation.** In a first step, the transformation computes the sequence of internal operations that need to be performed on the model without actually executing them. This set of changes is simply represented as a model called `Ma'-Ma`, which conforms to the `Diff` (for difference) metamodel. The `Ma'-Ma` model contains elements that represent the *create*, *update* and *delete* operations and their arguments.
- **Change application.** In a second step, operations are applied directly on the source model `Ma`, which results in the target model `Ma'`. Note that in this context, `Ma'` is actually a new version of `Ma`, not a separate model.

The notion of difference model can be also considered as analogous to the notion of difference file in the Unix world. In this analogy, the second step (i.e., applying changes) is similar to the `patch` command-line tool. However, in our case we do not compute changes between two source models. We compute the changes required to obtain the target model from a source model according to the transformation program.

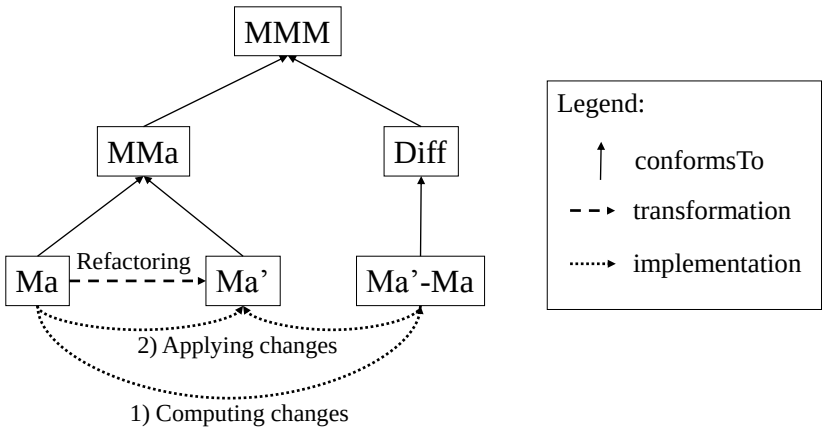

Figure 2: Overview of the in-place refining semantics.

It has to be noticed that also for the in-place case, the deletion of model elements needs a specific solution. While in the copy strategy, deletion could have been simulated by simply non-transforming some elements, the in-place semantics requires the implementation of an explicit deletion operation (non-transformed elements simply remain unchanged in the transformation output). The implementation details of this operation are dependent on the specific transformation language.

3.3 Comparison

Listing 2 shows that the copy strategy produces a dramatic improvement of the quality of the transformation code. However, the copy strategy does not provide any benefit in performance and integration, since it maintains the general transformation behavior unchanged (apart from the new default operation). In particular:

- The runtime performances are not better than the default mode. In some implementations they are even worse. This happens because the environment has to perform the same overall number of copy operations of the standard mode, with the additional burden of keeping trace of the matched elements and select the unmatched ones for the copying process.
- There is no guarantee about the preservation of identity information in the manipulated model, hampering any integration with model editors.

The in-place strategy maintains the same benefits on code quality than the copy strategy. For instance the code in Listing 2 will still be correct in in-place semantics. In-place mode even allows slightly shorter transformation rules, since the user is relieved from taking care of the correct copying of all the properties and references that are unchanged. For instance, lines 10, 11 and 13 could be omitted from Listing 2 in in-place mode, since these properties would just remain untouched.

More importantly, the in-place approach has two big advantages over the previous solutions:

- The possibility of significantly optimize the transformation execution time. The performance of the in-place strategy is generally better with respect to the copying one because it is not necessary to copy model elements of the unchanged part of the source model. When the target is saved in a different file than the source, this is the main benefit, because the serializer still has to traverse the target. When the model stays in memory, an additional benefit is that relatively large models may be processed without actually needing to traverse unchanged parts of the model. Of course, this is especially significant in the case where only small changes are performed on a very large model.
- A natural way of preserving identities for unchanged elements and links with external tools.

While these benefits raise the attractiveness of the in-place strategy for model refinement, they come at the cost of a significant development effort in implementing a new semantics. We will exemplify these implementation issues by describing in the next section the refinement support in ATL.

4 Refinement transformations in ATL

4.1 Copy strategy in ATL

So far, refinement transformations have been built in ATL using the copy strategy, by exploiting two technical means provided by the ATL environment: *rule superimposition* and *refining mode*.

4.1.1 Rule superimposition.

ATL provides a composition mechanism named *superimposition*. This is performed at runtime by weaving the bytecode of two transformations, so that the rules of the superimposed transformation override the rules with the same name in the other transformation. In [22] this mechanism has already been investigated for implementing refining transformation.

In the *Public2PrivateAndGetter* example, the user could generate automatically the copy rules for every element of the *ClassDiagram* metamodel (i.e. CopyClass, CopyDataType... and CopyAttribute). Then he would write a separate, superimposed transformation, containing only the rules related to the refactoring, i.e. PublicAttribute and CopyAttribute. The new CopyAttribute would override the generated one, avoiding to copy all the attributes and allowing the refactoring by PublicAttribute.

4.1.2 Refining mode.

The first explicit support for refining transformations in ATL has been added in 2005 in the form of a new transformation execution mode, and is called *refining mode*. This mode is selected in the ATL language by simply replacing the `from` keyword by the `refining` keyword in the transformation header. Obviously, refining mode may only be used for endogeneus transformations, i.e. when

source and target model share the same metamodel. Listing 3 shows how the header of a refactoring transformation on the ClassDiagram metamodel may look like.

Listing 3: ATL header for Public2PrivateAndGetter

```
1 module Public2PrivateAndGetter;
2 create OUT : ClassDiagram refining IN : ClassDiagram;
```

In refining mode the ATL transformation performs as if there were implicit copy rules that would match every element unmatched by explicit (i.e., programmer-written) rules. Such a copy rule creates a target element of the same type as the source element. Then, it initializes all the properties of the new element by copying the values of the properties of the source element. Additionally, it creates a traceability link. During the property initialization process, the ATL resolve algorithm [9] is used to find the target elements corresponding to source elements, navigating these trace links.

The refining mode implements the same algorithm of the traditional mode. The refining-specific behavior is triggered only when no traceability link exists for a source element submitted to the resolve algorithm. When this behavior is triggered, the source element is copied by calling a copy operation (called `refiningCopy`) added by the compiler into the generated bytecode. Pseudocode of this operation is presented in Listing 4.

Listing 4: Refining mode algorithm implemented in the refiningCopy operation

```
1  refiningCopy(sourceElement) {
2    If sourceElement is a Collection Then
3      ForEach e in sourceElement {
4        refiningCopy(e)
5      }
6    Else  -- sourceElement is a model element
7      targetElement <- create new element of same type as sourceElement
8      add trace link from sourceElement to targetElement
9      ForEach property of sourceElement {
10       v <- get value of property on sourceElement
11       set property of targetElement to resolve(v)
12     }
13   EndIf
14 }
```

The `refiningCopy` operation analyzes the type of the element to copy. If it is a collection, then a recursive call to `refiningCopy` is performed on every element (lines 3-5). If it is a model element, then a target element of the same type is created (line 7). Then, a loop iterates over the properties of the element (lines 9-12), and copies the values read from the source element to the target element. The resolve algorithm is called (line 11) before assigning the value to the target property. Note that this call to the resolve algorithm is also recursive because `refiningCopy` is only invoked by the resolve algorithm. This recursion is terminated if no more element are to be traversed. Note that if there is a loop in the model, then the resolve algorithm will not call `refiningCopy` again for the same element because the trace link has been created (at line 8) before the call to resolve (at line 11).

ATL refining mode provides a means for developers to avoid to transform part of the input model, without extending the ATL language with an explicit delete operation. Because `refiningCopy` is only called as part of the resolve algorithm, the parts of the model that are not connected to the elements transformed by explicit rules are ignored. This means that, if a developer wants to avoid

copying part of a model, he needs to write transformation rules that eliminate any reference to this part.

It has to be noticed, however, that, to make the mechanism more flexible, the algorithm for copy-based refining mode in the latest versions of ATL is slightly different than the one in Listing 4: for properties corresponding to composition associations, the value is actually copied only if the source element is the container. In other words, the copy of a container triggers the copy of its content but not viceversa, a design choice that mirrors the semantics of containment.

Summarizing, in the ATL copy strategy, elements are deleted if:

1. they are not copied by an explicit rule,
2. they are not contained in a copied element, and
3. they are not referenced (non-containement) by a copied element.

4.2 In-place strategy in ATL

In this section we introduce a novel semantics for ATL refining mode, natively in-place. This semantic is partially implemented in the latest released version of ATL. A prototype of the complete implementation is freely available on the AtlanMod wiki [2].

The semantics follows the scheme already introduced in Figure 2, and the Diff metamodel internally used by the ATL engine is expanded in Figure 3.

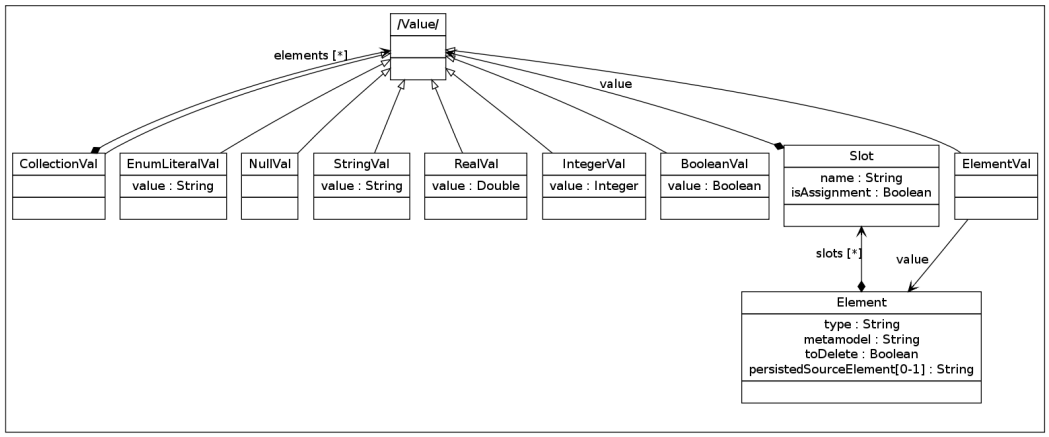

Figure 3: Diff Metamodel

The metamodel encodes the sequence of operations as a set of *Elements* to modify. An Element that does not point to a *persistedSourceElement* represents a create operation, and the Diff model encodes the property values of the new element as a set of *Slots*. Operations of *update* and *delete*

Listing 5: Private2Public.atl

```
1
2  rule PrivateAttribute {
3    from
4      s : ClassDiagram!Attribute (s.isPrivate and
5        s.owner.op->exists(o|o.name = 'get' +
6        s.name.toUpperCase() and o.returnType = s.type)
7        )
8    to
9      t : ClassDiagram ! Attribute (
10       isPrivate <- false
11     )
12 }
13
14
15 rule DeleteOperation {
16   from
17     s : ClassDiagram!Operation (s.owner.attr
18       ->exists(a|a.name = s.name.toUpperCase().substring(3, s.name->size())
19         and a.isPrivate))
20   to
21     drop
22 }
```

are encoded as Elements that point to a source element, and are distinguished thanks to a *toDelete* boolean attribute.

In the first step, the transformation engine executes the rules and fills the Diff model with calculated changes. Then the basic operations are reordered and applied. The reordering is possible because in standard ATL every *create* and *update* operation is independent, since source elements are matched just once (other languages could require in this phase a different algorithm, sometimes with conflict resolution).

A *delete* instead can easily generate a conflict with another operation. This happens because we chose to provide *delete* with a cascade-delete semantics for composition associations (i.e. the deletion of a container triggers the deletion of the contained elements). For this reason the user is allowed to specify that he wants to create a new contained element, and, at the same time, that he wants to delete the container. In this case, and in similar ones, we solve the conflict introducing a default policy that gives to *delete* a higher priority.

The whole Diff-application algorithm is implemented by first applying all the *create* operations, then all the *modify* operations, and finally all the *delete* operations. It's easy to verify that this reordering satisfies the policy that we described for conflict resolution.

Finally, in our prototype implementation we decided to also extend the ATL syntax, by introducing an ad-hoc keyword for deletion. This step was not strictly necessary, but 1) it provides the user with a more readable code for a destructive operation like *delete* and 2) it allows to mix in the same rule deletions and other operations. The ATL abstract syntax has been modified, by adding a new statement, called *drop*, to the ATL metamodel, and the concrete syntax has been updated by extending the parser to recognize the DROP keyword. In the listing 5, we show how to use the new syntax to undo the changes introduced by the motivating example.

Table 1: Public2Private.atl on j2se-1_2-api.uml (6.17MiB)

	Time (s)		Bytecodes	Lines of code
	Transformation	Total		
Copy	209	216	24677797	41
In-place	3.20	11	354364	15
Ratio	65.3	19.6	69.6	2.7

Table 2: Public2Private.atl on j2se-1_6-api.uml (13.14MiB)

	Time (s)		Bytecodes	Lines of code
	Transformation	Total		
Copy	496	515	52478984	41
In-place	5.75	25	674874	15
Ratio	86.3	20.6	77.8	2.7

4.3 Comparison

Tables 1 and 2 give the results of a comparative benchmark of two implementations of transformation *Public2Private*: one based on default-copy by superimposition and the other based on in-place refining. Both implementations of the transformations have been executed on large UML models obtained by reverse engineering the Java API (versions 1.2 for Table 1, and version 1.6 for Table 2). The first column gives the transformation execution time, without taking into account model loading and saving. Total time is given in the second column. Note that transformation time is highly relevant in many cases like: in the middle of a chain, when refactoring a model in memory (e.g., in a CASE tool). The total number of executed bytecodes is given in the third column, while the fourth column gives the number of lines in the source program.

The execution time of in-place mode results remarkably lower than the solution based on superimposition. We point out that the third possible implementation, using the default-copy refining mode, would be even slower than superimposition. The reason is that this refining mode uses reflection to perform the copy whereas explicit copy rules directly access and create elements.

5 Related work

Techniques for efficiently implementing refinement rules have been extensively studied in the Program Transformation field. Examples of transformations languages in this area, with a direct support for in-place transformations are Stratego [21], ASF+SDF [18] and TXL [5]. A related survey of possible strategies can be found in [20]. Our proposal takes inspiration by these works to study the particular problem of adapting pre-existing M2M frameworks to model refinement.

Since the importance of model refinement has been recognized in several MDE approaches, like OMG's MDA[13], other model transformation languages have been adapted to enable it (e.g. [23]).

In [8] the authors add support for default copy rules in QVT. The approach has some similarities with our default-copy refinement semantics, and is implemented by an higher-order transformation.

In the Epsilon family of languages, the main transformation language, ETL[11], does not have a special support for refining transformations (that are called "update transformations in the small"). The family comprises a different language, Epsilon Wizard Language [12], specifically designed for this purpose. In this paper we want to help transformation language designers. We propose to do that by implementing a specific refinement support to their M2M languages, instead of developing a new ad-hoc language.

In this work we consider only the technical aspect of refinement and refactoring, as pure manipulations of the abstract syntax. In actual applications, semantic aspects of the refinement transformation have to be taken into account. However, this problem is beyond our scope and we refer the reader to works on semantics of model-to-model transformations ([4], [16]). For instance, several works target the semantic preservation of a designer-defined refactoring transformation (for example [10]) to ensure its correctness.

Finally, while the approach we are following investigates the application of a language with a natively separated input and output models to the in-place case, some works in the graph transformation area follow the opposite path. [15] studies how to apply a natively in-place transformation language to efficiently describe translations.

6 Conclusions and future work

In this paper, we have considered the problem of expressing refining transformations using model-to-model transformation languages initially designed for translational transformations. More specifically, we focus on rule-based M2M transformation languages, that have read-only source model and write-only target model semantics. The paper discusses two main strategies for facilitating refinement transformations (i.e., refining mode) in this kind of model transformational languages : 1) implicit copy of unmatched elements from source to target, or 2) in-place modifications of the source model, resulting in the target model. Trade-offs of both strategies are compared against a set of dimensions like code quality, performance, integrability. An implementation of both strategies in ATL is described.

In future work we want to improve our in-place refinement engine, by adding native support for advanced operations on models, to complement our create-update-delete scheme. Examples of the operations that we are evaluating are *transtype* (i.e. changing the type of a model element without changing its identity), *redirect* (i.e. reroute all the references pointing to an element to another one) and *clone* (i.e. making multiple copies of an element).

We are also going to extend and formalize our comparison of M2M and graph transformation approaches. The intention is to evaluate how these different families of transformation technologies relate to each other in terms of expressivity, complexity and performance.

References

[1] A. Agrawal, G. Karsai, S. Neema, F. Shi, and A. Vizhanyo. The design of a language for model transformations. *Software & Systems Modeling*, 5(3):261–288, July 2006.

[2] AtlanMod. http://www.emn.fr/z-info/atlanmod/index.php/ATL_Refining_Mode - ATL Inplace Refining Mode, 2011.

[3] J. Bézivin, F. Jouault, G. Pitette, G. Dupé, and J. E. Rougui. First experiments with the ATL model transformation language: Transforming XSLT into XQuery. In *2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture*. Springer, 2003.

[4] E. Biermann, C. Ermel, and G. Taentzer. Precise Semantics of EMF Model Transformations by Graph Transformation. In *Proceedings of the 11th International Conference, MoDELS 2008*, pages 1–15. Springer, 2008.

[5] J. Cordy. The TXL source transformation language. *Science of Computer Programming*, 61(3):190–210, Aug. 2006.

[6] Eclipse M2M. http://www.eclipse.org/m2m/atl/atlTransformations/ - ATL Transformation Zoo, 2011.

[7] T. Fischer, J. Niere, L. Torunski, and A. Zundorf. Story diagrams: A new graph rewrite language based on the unified modeling language and java. *Theory and Application of Graph Transformations*, pages 157–167, 2000.

[8] T. Goldschmidt and G. Wachsmuth. Refinement Transformation Support for QVT Relational Transformations. In *3rd Workshop on Model Driven Software Engineering (MDSE 2008)*. Springer, 2008.

[9] F. Jouault and I. Kurtev. Transforming Models with ATL. In *Proc. of the Model Transformations in Practice Workshop at MoDELS 2005*, volume Satellite, pages 128–138. Springer, 2005.

[10] S. R. Judson, D. L. Carver, and R. France. A Metamodeling Approach to Model Refactoring - Technical Report, 2003.

[11] D. Kolovos, R. Paige, and F. Polack. The epsilon transformation language. *Theory and Practice of Model Transformations*, pages 46–60, 2008.

[12] D. S. Kolovos, R. F. Paige, F. A. Polac, and L. M. Rose. Update Transformations in the Small with the Epsilon Wizard Language. *The Journal of Object Technology*, 6(9):53, 2007.

[13] S. Mellor, K. Scott, A. Uhl, and D. Weise. Model-Driven Architecture. In *Advances in Object-Oriented Information Systems*, volume 2426 of *Lecture Notes in Computer Science*, pages 233–239. Springer, Sept. 2002.

[14] Object Management Group. QVT 1.0 - http://www.omg.org/spec/QVT/1.0/, 2008.

[15] A. Schurr. Specification of graph translators with triple graph grammars. In *Graph-Theoretic Concepts in Computer Science*, pages 151–163. Springer, 1995.

[16] R. V. D. Straeten, V. Jonckers, and T. Mens. A Formal Approach to Model Refactoring and Model Refinement. *Software and Systems Modeling - Springer*, 2006.

[17] M. Tisi, F. Jouault, P. Fraternali, S. Ceri, and J. Bézivin. On the Use of Higher-Order Model Transformations. In *Proceedings of the Fifth European Conference on Model-Driven Architecture Foundations and Applications (ECMDA)*, pages 18–33. Springer, 2009.

[18] M. G. J. Van Den Brand, P. Klint, and J. J. Vinju. Term rewriting with traversal functions. *ACM Transactions on Software Engineering and Methodology*, 12(2):152–190, Apr. 2003.

[19] D. Varró, G. Varró, and A. Pataricza. Designing the automatic transformation of visual languages. *Science of Computer Programming*, 44(2):205–227, Aug. 2002.

[20] E. Visser. A survey of strategies in rule-based program transformation systems. *Journal of Symbolic Computation*, 40(1):831–873, July 2005.

[21] E. Visser, Z.-e.-A. Benaissa, and A. Tolmach. *Building program optimizers with rewriting strategies*. ACM Press, New York, New York, USA, 1998.

[22] D. Wagelaar. Composition techniques for rule-based model transformation languages. *Theory and Practice of Model Transformations*, pages 152–167, 2008.

[23] J. Zhang, Y. Lin, and J. Gray. Generic and Domain-Specific Model Refactoring using a Model Transformation Engine. In *Conference Proceedings of the Models Conference*. Springer, 2005.

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399