Towards a traceability framework for model transformations in Kermeta

Jean-Rémy Falleri, Marianne Huchard, and Clémentine Nebut

LIRMM, CNRS and Université de Montpellier 2,
161, rue Ada, 34392 Montpellier cedex 5, France
{falleri, huchard, nebut}@lirmm.fr

Abstract. Implementing a model transformation is a very complex task and in an MDA process, chains of model transformations are usually built. When writing such a transformation chain, developers often need to have information on the previously applied transformations. Thus, disposing of a traceability framework enabling to gather information on the transformation behavior is an important feature for a transformation language. In this paper, we propose to implement a traceability framework in the Kermeta language based on a language independent trace metamodel.

1 Introduction

In the model-oriented paradigm proned by the MDA [1], model transformations [2] are of crucial importance. Since model transformations can be very complex, and since source and target models can be composed of a great number of elements, it is very difficult for the developer and the user to figure out exactly how a transformation behaves. An MDA process generally involves several successive transformations, and if we consider such a chain of transformations, it is almost impossible, even for the developer, to recognize the elements in the first model that have led to the generation of a given element in the last model of the chain.

Moreover, as explained in [3], some model transformations can be performed only when they dispose of the trace of anterior transformations. Another good example of the utility of traceability information is a refactoring chain, where you transform a model conform to a metamodel into an improved model conform to the same metamodel, through a chain of transformations (for example, UML class hierarchy restructuring using Formal Concept Analysis [4]). In this kind of transformation chain, a lot of information of the source model is lost during the chain but is expected to be reinjected in the target model. Thus, developers need a system to gather information on the transformation behaviour when executing it. Such a system is usually called a traceability framework.

In this paper, we propose a simple traceability framework that is sufficient to solve the previous issues. This framework is based on a model definition (model as a set) inspired by [5], which allows a basic trace metamodel to be defined. This framework is implemented in the model oriented language Kermeta [6]. Model transformations can be defined using Kermeta [7], but they are difficult to trace due to the imperative syntax of the language.

The paper is structured as follows. Section 2 presents our approach. The Kermeta implementation and its usage are presented in Sections 3 and 4. Section 5 discusses the benefits and limitations of this approach, and related work as well.

2 An intuitive model definition

To trace a model transformation, two concepts have to be precisely defined: what is a model and what is a model transformation. To remain in the model paradigm described by the MDA, these two concepts should be described by a model. Several metamodels are available to represent what is a model (MOF [8] for instance), but there is still a lack of consensus to exactly describe what is a model transformation. The main rationale for this lack is the difficulty to write a transformation metamodel independent from the transformation language. Anyway, if we want to trace model transformations, disposing of such a model is fully necessary. We need to know what are the basic entities of a model transformation, if we want to be capable of storing and analysing them. To obtain a trace metamodel independent from the transformation language, a simple and efficient method to model a model is to see it as a set, which is composed of elements (as shown in Figure 1).

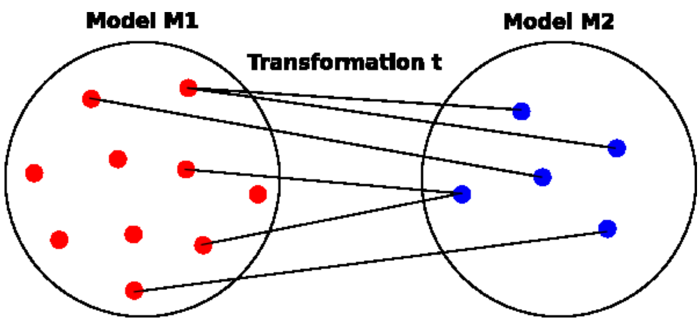

Fig. 1. Model transformation

Definition 1. *A model M is a set of elements.*

Definition 1 leads to the following definition of model transformations:

Definition 2. *Let M1 and M2 be two models. A model transformation is a relation* $t, t \subseteq M1 \times M2$*.*

Note in Figure 1 that an element of a source model can be linked to n elements of the target model, or n elements of the source model can be linked to one element of the target model. Moreover, every target element has a parent. Based on the previous definition of a model transformation, we propose the following definition for a model transformation trace.

Definition 3. *A transformation trace is a bipartite graph. The nodes are partitioned into two categories: source nodes and target nodes.*

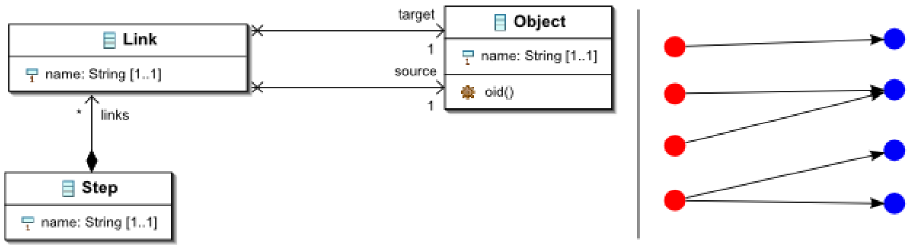

Fig. 2. Transformation trace metamodel and model

Therefore, if we want to be able to keep a trace of a model transformation, we have to set up the metamodel of a bipartite graph structure. As shown in Figure 2, a trace *Step* is composed of several *Link*. A *Link* references two *Object*: the source one and the target one. *Object* is the most general kind of element that can be found in the *Kermeta* language. Thus, it ensures that every type of element will be storable in the trace. This metamodel was designed to store the trace of a single transformation, and that is the reason why the related element has been named *Step*. But since transformations are usually small units which can be chained, we propose to modify this first metamodel to integrate as best as we can this usage of model transformations.

Let's consider now a transformation chain trace. This kind of trace is slightly different from the one above. It is an ordered set of bipartite graphs with a common intersection (some target nodes of a graph are source nodes of the following graph). To handle that type of trace, we have extended our previous metamodel by adding a *Trace* object, which contains an ordered set of *Step*. The result of this extension is shown in the Figure 3.

Now that we have determined what is a trace, we can define some operations on it. Let's consider the trace shown in the Figure 4.

Definition 4. *The direct parents of an element are the elements which are directly linked to it. For instance,* $parents(C3) = \{B3, B4\}$.

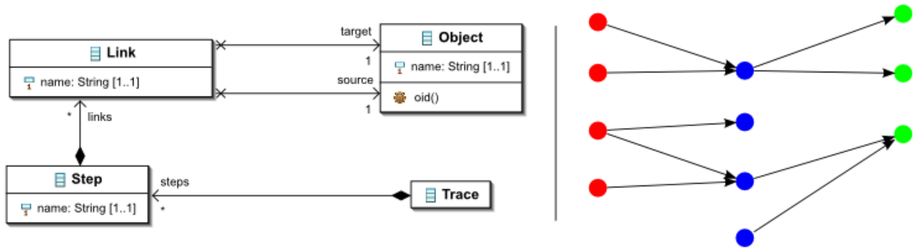

Fig. 3. Transformation chain trace metamodel and model

Definition 5. *The parents of an element are the direct parents of the element and the direct parents of the direct parents (recursively). For instance,* $allparents(C3) = \{B3, B4, A4, A3\}$.

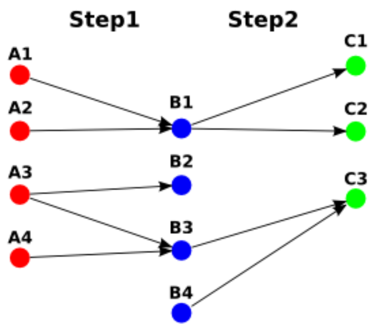

Fig. 4. A sample transformation chain trace

3 Kermeta implementation

The mechanisms to easily handle a trace conform to the previously defined trace metamodel have been implemented in Kermeta [6]. Kermeta is a model oriented language which allows to define metamodels and to give them semantics. Moreover, it is fully compatible with the Eclipse implementation of EMOF: EMF [9]. We have implemented the following features in the traceability framework:

1. Generic traceability items;
2. Trace serialization (in XMI 2.0, thanks to EMF);
3. Simple transformation from a trace to graphviz's [10] dot language, in order to allow trace vizualisation.

We have also respected the following constraints:

1. Trace generating code should be as short as possible, and only a small part of it should be placed in the transformation code;
2. Developers must be able to access to the elements of their choice through the trace;
3. Developers must be able to select the elements they want to trace.

We can notice that Feature 2 and Constraint 2 are in contradiction. Indeed, if we want to serialize the trace, all the *Object* should be contained in it (otherwise we will only serialize reference, which is not really interesting). But if we want to modify a real model element, *Object* should only be references to real elements. That is why we have setted up two kinds of traces: *Trace* and *StaticTrace*. *Trace* contains only references and is used in the transformations, whereas *StaticTrace* contains the *Object* and can be serialized. To make the link between these two kinds of trace, a transformation from a *Trace* to a *StaticTrace* has been written. Listing 1.1 shows an extract of the trace metamodel source code implemented in the framework. Listing 1.2 shows the metamodel used to serialize the trace. Before being stored in the *StaticTrace*, the elements are transformed into a reduced form, named *Element* which contains only the information needed to allow a readable visualization.

Listing 1.1. Transformation trace metamodel

```
class Trace
{
    attribute steps: oset Step[0..*] // oset: Ordered Set

    /**
     * Adds a link in the given step between the two given
         objects
     */
    operation add_link(step: String, name: String, source:
        Object, target: Object): Void is do end

    /**
     * Inits a new step with the given name
     */
    operation init_step(name: String): Void is do end

    /**
     * Returns the direct parents of the given object
     */
    operation parents(target: Object): Set<Object> is do end

    /**
     * Returns all parents of the given object
     */
    operation all_parents(target: Object): Set<Object> is do
        end
```

```
    /**
     * Returns all parents in the given step of the given
         object
     */
    operation parents_at_step(target: Object, step: String):
        Set<Object> is do end
}

class Step
{
    attribute name: String[1..1]
    attribute links: Link[0..*]

    /**
     * Adds a link with the given name between the two given
         objects
     */
    operation add_link(name: String, source: Object, target:
        Object) is do end
}

class Link
{
    attribute name: String[1..1]
    reference source: Object[1..1]
    reference target: Object[1..1]
}
```

Listing 1.2. Static transformation trace metamodel

```
class StaticTrace
{
    attribute elements: Element[0..*]
    attribute steps: oset StaticStep[0..*] // oset: Ordered
        Set
}

class StaticStep
{
    attribute name: String[1..1]
    attribute links: StaticLink[0..*]
}

class StaticLink
{
    attribute name: String[1..1]
    reference source: Element[1..1]
    reference target: Element[1..1]
}
```

```
class Element
{
    attribute id: String
    attribute label: String
}
```

In order to manipulate the traces produced, an utility class has been written. The code of this class can be found in Listing 1.3. The only thing remaining to be done before being able to generate some dot code or to serialize a trace is to specialize the *TraceUtils* class, in order to implement the abstract *label(object: Object): String* operation. It ensures that the label of each *Element* generated will be understandable.

Listing 1.3. Utility class designed to manipulate trace

```
class TraceUtils
{
    /**
     * Prints the dot code reprensenting the given Trace
     */
    operation trace_2_dot(trace: Trace) is do end
    /**
     * Converts a Trace into a StaticTrace
     */
    operation trace_2_static_trace(trace: Trace): StaticTrace
        is do end

    /**
     *  Prints the dot code reprensenting the given
         StaticTrace
     */
    operation static_trace_2_dot(trace: StaticTrace): Void is
        do end

    /**
     * Saves the given StaticTrace to the given filename
     */
    operation save_static_trace(trace: StaticTrace,filename:
        String): Void is do end

    /**
     * Loads the StaticTrace from the given filename
     */
    operation load_static_trace(filename: String): StaticTrace
         is do end

    /**
```

```
     * Abstract labelling operation used in the trace_2_dot
        and the trace_2_static_trace operations
     */
    operation label(object: Object):String is abstract
}
```

4 The traceability framework in action

To illustrate how this package works, we use a very simple transformation example: a class hierarchy (as defined by the *minuml* metamodel) will be turned into a database (as defined by the *mindb* metamodel). The two metamodels involved in the transformation are given in Figure 5.

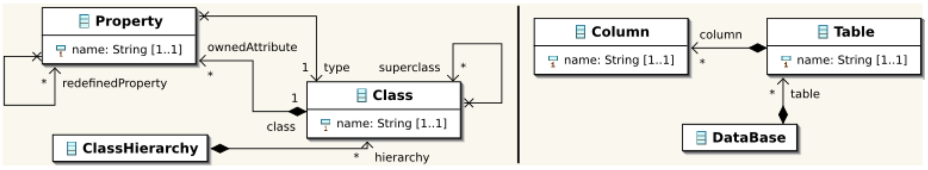

Fig. 5. *minuml* and *mindb* metamodels

As it can be noticed in the *minuml* metamodel, a *ClassHierarchy* is composed of some *Class*. These *Class* have some *Property*. A *Property* has a *name*. The *mindb* metamodel has a similar structure as the *minuml* one.

With these two metamodels, it is now possible to write a standard transformation between a *ClassHierarchy* and a *DataBase* (the code is given in Listing 1.4). This transformation is very simple: a *Table* is created for each *Class* of the source model, and a *Column* is added in the *Table* for each *Property* of the *Class*.

Listing 1.4. *Minuml2Mindb* transformation code

```
/**
 * Transform a minuml model to a mindb model
 */
operation transform(source: ClassHierarchy): DataBase is do
    result := DataBase.new // Initialize the target model

    trace.initStep("minuml2mindb") // Trace Generating Code

    source.hierarchy.each{ cls | // Iterate on every class of
        the source model
        var table: Table init Table.new // Create a Table
        table.name := String.clone(cls.name) // Copy the name
            of the Class to the table
```

```
        result.table.add(table) // Add the table in the target
             model

        trace.add_link("minuml2mindb","class2table",cls,table)
             // Trace Generating Code

        cls.ownedAttribute.each{ prop | // Iterate on every
            Property of the Class
            var col: Column init Column.new // Create a new
                Column
            col.name := String.clone(prop.name)
            table.column.add(col) // Add the Column to the
                relative Table

            trace.add_link("minuml2mindb","property2column",
                prop,col) // Trace Generating Code

        } // End Iterate on every Property of the Class
    } // End Iterate on every class of the source model
end
```

Figure 6 shows the generated dot graph from the following transformation chain: *minuml* → *mindb* → *minuml* applied to a sample *minuml* model.

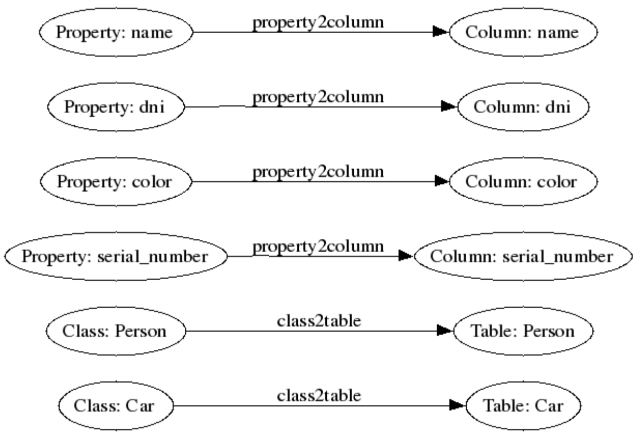

Fig. 6. The generated transformation trace

5 Perspectives and conclusion

With the framework we proposed in this paper, it is possible to trace transformations within Kermeta. But it is still possible to improve the *Trace* metamodel and thus the framework. The first thing that is currently studied is the possibility to add composite links in a *Step*, which will be defined as a set of *Link*. A composite link would be associated to a significative part of a transformation (it could gather for instance all links relative to column creation of a given table). It would allow to add more semantics in the trace.

When implementing this framework, it was very clear that the way used to handle the trace is very dependent from the choosen implementation language. As shown in the previous section, using our traceability framework requires to add trace generation code in the transformation code. With an imperative language, such code is unavoidable even if not wished (traceability should be as transparent as possible). To have trace generation code easily written and identified in a transformation, several solutions can be found, among them adding tags dedicated to the traceability management in the Kermeta language.

Last, seeing models as sets leads to a simple and efficient definition of a trace, that is sufficient in many situations. Since models are not simply sets but rather directed and labeled graphs, it would be useful to construct a trace metamodel based on this definition. With such a trace, it would be possible to store more information than with the previous one. It would show for instance the composition relation between a table and some columns.

References

1. OMG: MDA guide version 1.0.1. http://www.omg.org/docs/omg/03-06-01.pdf (2003)
2. Czarnecki, K., Helsen, S.: Classification of model transformation approaches. In: OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture, Proceedings. (2003)
3. Vanhooff, B., Berbers, Y.: Supporting modular transformation units with precise transformation traceability metadata. In: ECMDA-TW 2005 Proceedings, Nuremberg, November 8th 2005. (2005)
4. Dao, M., Huchard, M., Hacene, M.R., Roume, C., Valtchev, P.: Improving Generalization Level in UML Models Iterative Cross Generalization in Practice. In: International Conference on Conceptual Structures ICCS'04. (2004)
5. Jouault, F.: Loosely coupled traceability for atl. In: ECMDA-TW 2005 Proceedings, Nuremberg, November 8th 2005. (2005)
6. Triskell project (IRISA): The metamodeling language kermeta. http://www.kermeta.org (2006)
7. Muller, P.A., Fleurey, F., Vojtisek, D., Drey, Z., Pollet, D., Fondement, F., Studer, P., , Jézéquel, J.M.: On executable meta-languages applied to model transformations. In: Model Transformations In Practice Workshop. Proceeding, Montego Bay, Jamaica. (2005)
8. OMG: MOF 2.0 core specification. http://www.omg.org/cgi-bin/doc?ptc/2004-10-15 (2004)
9. Eclipse: The eclipse modeling framework (EMF) overview. http://www.eclipse.org/emf/docs.php?doc=references/overview/EMF.html (2005)
10. Graphviz: Graph visualization software. http://www.graphviz.org (2001)