

# Traceability as Input for Model Transformations <sup>\*</sup>

Bert Vanhooff, Stefan Van Baelen, Wouter Joosen, and Yolande Berbers

Department of Computer Science, K.U.Leuven, Celestijnenlaan 200A, 3001 Leuven, Belgium  
{bert.vanhooff, stefan.vanbaelen, wouter.joosen,  
yolande.berbers}@cs.kuleuven.be

**Abstract.** Some model transformations require more information than can be derived from its source model(s) in order to generate a meaningful target model. For example, a transformation with two source models needs to know how their respective model elements relate; these relations often only exist implicitly as part of the transformations developer’s knowledge. In this paper we show that traceability models, who can be automatically generated as part of any model transformation, contain explicit inter- and intra-model relations that are valuable to subsequent transformations. We explain how to extract this information and propose a number of additions to current transformation techniques that are needed to completely open up traceability information to transformation developers.

## 1 Introduction

Classically, the creation and maintenance of traceability information has been a manual and labor intensive task. The rise of Model Driven Development (MDD) eases this problem by introducing model transformations, which can generate basic traces automatically. MDD recognizes the need of many different intermediate models to represent a system, which results in an abundance of readily available traces that interrelate every piece of the system. Typical uses of this traceability information are: showing that requirements are met, analyzing impact of changes, propagating changes, etc.

In this paper we present a new use of traceability information in the context of transformation chains. A transformation chain or transformation composition is a network of many subtransformations, each contributing a small part to a larger transformation goal. Each subtransformation also produces traces that contribute to a *global traceability graph*. This graph interrelates disparate models that represent different aspects of a system (e.g. functional, security and persistence model), produced as part of a transformation chain. In some cases we need to know exactly how these aspects are related in order to perform further transformations (e.g. which database table stores a particular attribute from the functional model). This kind of information usually cannot be extracted from the individual models but is hidden in the traceability models.

We will show how generated traces can provide non-trivial inter- and intra-model relations without imposing many additional requirements on the models or transformations. We define *trace navigation* as an operation to derive the required information

---

<sup>\*</sup> The described work is part of the EUREKA-ITEA MARTES project, and partly funded by the Flemish government institution IWT (Institute for the Promotion of Innovation by Science and Technology in Flanders).

and propose *trace tagging* to enrich the information contained in the traceability graph. Finally we discuss how these concepts can be integrated in current transformation technologies.

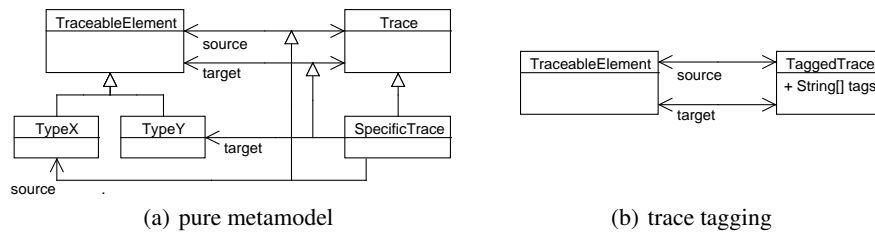
The rest of the paper is structured as follows. Section 2 provides the necessary background on traceability modeling and trace generation. In Section 3, we show that generic traceability information can be used to derive non-trivial relations between model elements. In some cases, we must add a specific meaning to traces in order to extract useful information (Section 4). In Section 5 we discuss how our approach can be integrated in current transformation technologies. We wrap up by presenting related work (Section 6) and drawing conclusions (Section 7).

## 2 Background

In the context of MDD, traceability information is represented as a model on its own. In this section we briefly discuss what a traceability (meta)model is (2.1) and how traces can be generated automatically by transformations (2.2).

### 2.1 Traceability Metamodel

A traceability metamodel captures how model elements may be related by trace relations and defines the semantics of such relations. It is often necessary to distinguish between different kinds of traces. For example, it is interesting to know whether a trace denotes a relationship between a textual requirement and an architectural element or indicates a refinement within the design. The required kinds of traces are often highly project dependent [1]. In this subsection, we make a concise comparison between two different traceability metamodels that both facilitate specific kinds of traces.



**Fig. 1.** Two extreme traceability metamodels.

In a *pure metamodel* approach, the traceability metamodel specifies every kind of trace we may need. Figure 1(a) shows a snapshot of such a metamodel. The two top classes indicate that a *Trace* can relate two *TraceableElements*. If we need a more specific kind of trace it suffices to subclass *Trace* and specialize its associations (see *SpecificTrace* in the figure). The most notable advantage of this approach is that we can precisely specify both usage constraints and semantics of each trace kind at the metamodel

level. The main drawback is its lack of flexibility: every change in traceability requirements needs to be reflected (hard-coded) in the metamodel. Project-specific trace kinds are expressed directly in the metamodel, which limits potential reuse to one project.

The *trace tagging* approach uses a simple and general traceability metamodel (see Figure 1(b)) and allows users to annotate generic traces with attributes or *tags*. A similar approach is used in specialized tools such as Telelogic DOORS. The kinds of traces are represented by tags and are defined at the model rather than at the metamodel level. This prevents us from specifying precise semantics and usage rules in the metamodel. A user can add any tag without having to adhere to any rules. At the same time this kind of flexibility is its biggest advantage. The metamodel never needs to be changed, hence this kind of metamodel can be reused in any project.

We will show that, because of its generality and flexibility, the trace tagging approach is best suited for integrating traceability into transformation compositions.

## 2.2 Automatic Trace Generation

Automatic model transformations can generate traceability information along with the target model(s). According to [2], transformation approaches either have dedicated support for traceability or rely on the developer to encode traceability as a regular output model. In any case, it is favorable to incorporate traceability generation into the transformation as opposed to producing it manually, no matter whether we apply the pure metamodel or tagging approach.

The major advantage of dedicated traceability support is that we get the traceability model(s) at an extremely low cost; the developer has to do little to no additional effort. A practical disadvantage is that the traceability metamodel is then fixed and may not be standardized among different transformation engines. Also, the level of granularity of the traces may not be the same.

Alternatively we can treat traceability as a regular output model of the transformation and incorporate additional transformation rules to generate it. The choice of metamodel is then completely at the discretion of the developer and does not depend on the transformation engine. The drawback is that additional effort is required to add traceability-specific transformation rules, which also pollute the implementation. An approach that partly solves these issues by automatically generating the trace-related transformation rules was proposed in [3].

Since our research is focussed on composing many subtransformations and combining different transformation technologies, we can only assume a generic traceability metamodel (i.e. Figure 1(b)) as the least common denominator of all transformation technologies. Furthermore we assume that each transformation creates a separate trace for each generated target model element; this link points to the source element(s) from which the target was created. Hence, no model element exists that is not fully traceable back to its source throughout the transformation chain.

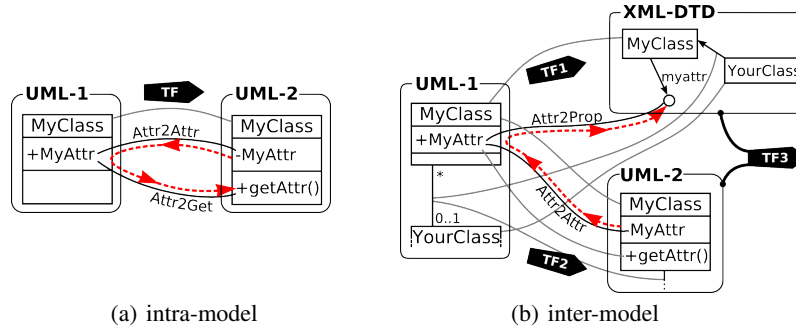
## 3 Leveraging Traceability Models

In an MDD approach we use many models that offer different views on a system: different levels of abstraction, different sub-aspects of a system, etc. Many of these models

are derived through automatic transformations and are hence highly interlinked through automatically generated traceability models (as discussed in 2.2). In 3.1 we show that the information contained in traceability models can be leveraged for development of further transformations. We discuss an elaborate example in subsection 3.2.

### 3.1 Discovering Useful Trace Information

Figure 2(a) and 2(b) present examples of transformations and models along with their generated traceability models. Each curved line in the figure represents a single trace and is part of a traceability model. For example in Figure 2(a) there are in fact three models: UML-1, UML-2 and the traceability model that connects these two.



**Fig. 2.** Deriving model relations from past traceability information.

Figure 2(a) shows a transformation **TF** from source **UML-1** to target **UML-2** that adds accessor methods (i.e. getters) to classes and makes corresponding attributes private. We focus on traces **Attr2Attr** and **Attr2Get**, labeled according to their transformation rule. The dotted arrows on the figure indicate that we can navigate back and forth through the traceability model and the **UML-1** model to discover an *intra-model* relation between **MyAttr** and **getAttr()** in the **UML-2** model. A subsequent transformation can use this information, which matches attributes to their accessors, for example to generate a detailed implementation for accessors. As a result, the concern of naming the accessor and making the attribute private (transformation **TF**) can be separated from the implementation generation. The necessary information is implicitly passed via the generated traces. Without traceability, a subsequent transformation would only be able to guess the relation between accessor and attribute (e.g. by matching strings).

A second example is shown in Figure 2(b). Class model **UML-1** is transformed into an XML data type definition model (**XML-DTD**) [4] and a refined class model **UML-2**. This transformation is accomplished in two parallel steps (**TF1** and **TF2**); **TF2** is equivalent to the transformation in Figure 2(a). Suppose that we now introduce an additional subsequent transformation (**TF3**) to generate a persistence layer that can load objects from an XML file. This transformation needs to know, amongst others, the exact relations between class attributes and XML properties in order to produce a valid target

model. If we solely use UML-2 and XML-DTD as inputs it is hard to find these relations. Resorting to the traceability models offers a solution. For example, the XML property that corresponds to class attribute *MyAttr* can be found by navigating back (*Attr2Attr*) and forth (*Attr2Prop*) through the traceability model (see dotted arrows). This example shows that we can also derive *inter-model* relations from traceability models.

In the former paragraphs we have used ‘navigate’ whenever we combined the information from one or more traces to derive relations between transformation input model elements. We consider the union of all traceability models and regular models as a graph  $G$  where elements from the regular models are vertices  $V$  and traces are edges  $E$ . We can hence give a definition for trace navigation.

**Definition 1.** *Given an input vertex  $v_i$  and a set of navigation target models  $M_t$ , a **trace navigation** is any operation that takes  $v_i$  as input and yields a collection of vertices  $V_o$  as output. All the elements of  $V_o$  are reachable from  $v_i$  and are owned by a model in  $M_t$ .*

### 3.2 Example: Persistence with a Relational Database

In this subsection we present a realistic example where a relational persistence layer is generated by a composition of a transformations. The complete composition is shown in Figure 3.

In the upper part,  $TF_{A1}$  and  $TF_{A2}$  subsequently transform the initial class model into an entity-relationship model (ER) [5] and corresponding relational database tables (RDB/SQL). Notice that the transformation from entity-relationship to database tables ( $TF_{A2}$ ) is not a trivial one-to-one mapping; only two tables are introduced to represent three elements of the ER model. In the lower part,  $TF_{B1}$  and  $TF_{B2}$  transform the class model into a simplified class model UML-2 (taking away the association class) and subsequently a JAVA model. The figure also shows all the traces that were created by each subtransformation.

Having this many intermediate models has the advantage that different concerns are treated separately in different transformations and are visible in separate, highly specialized models. This narrows the scope of each subtransformation, making them easier to implement, and offers a dedicated view to each domain expert – i.e. a database specialist considers the ER model while a Java programmer only looks at the JAVA model.

After executing all the transformations, we end up with the *current* models: RDB/SQL and JAVA. At this point we introduce a new transformation  $TF_{NEW}$  (see Figure 3) that adds database access code to the JAVA model; part of the resulting model is shown in the rightmost part of the figure. Transformation  $TF_{NEW}$  takes both RDB/SQL and JAVA models as input since a persistence layer involves getting values from the tables in the database (RDB/SQL model) and storing them as class attribute values of the JAVA model.

In Figure 3 we show how we can relate classes to database tables in order to generate the correct sql query for the *Person* class. Just by navigating the traces through the old models, we end up with the *Employee* table (follow the lines marked with dots), which is indeed the table where the information about *Person* is stored; note that we also get the corresponding primary key. It would be very difficult to find this relation without the traceability information since there is no one-to-one correspondence from the RDB/SQL

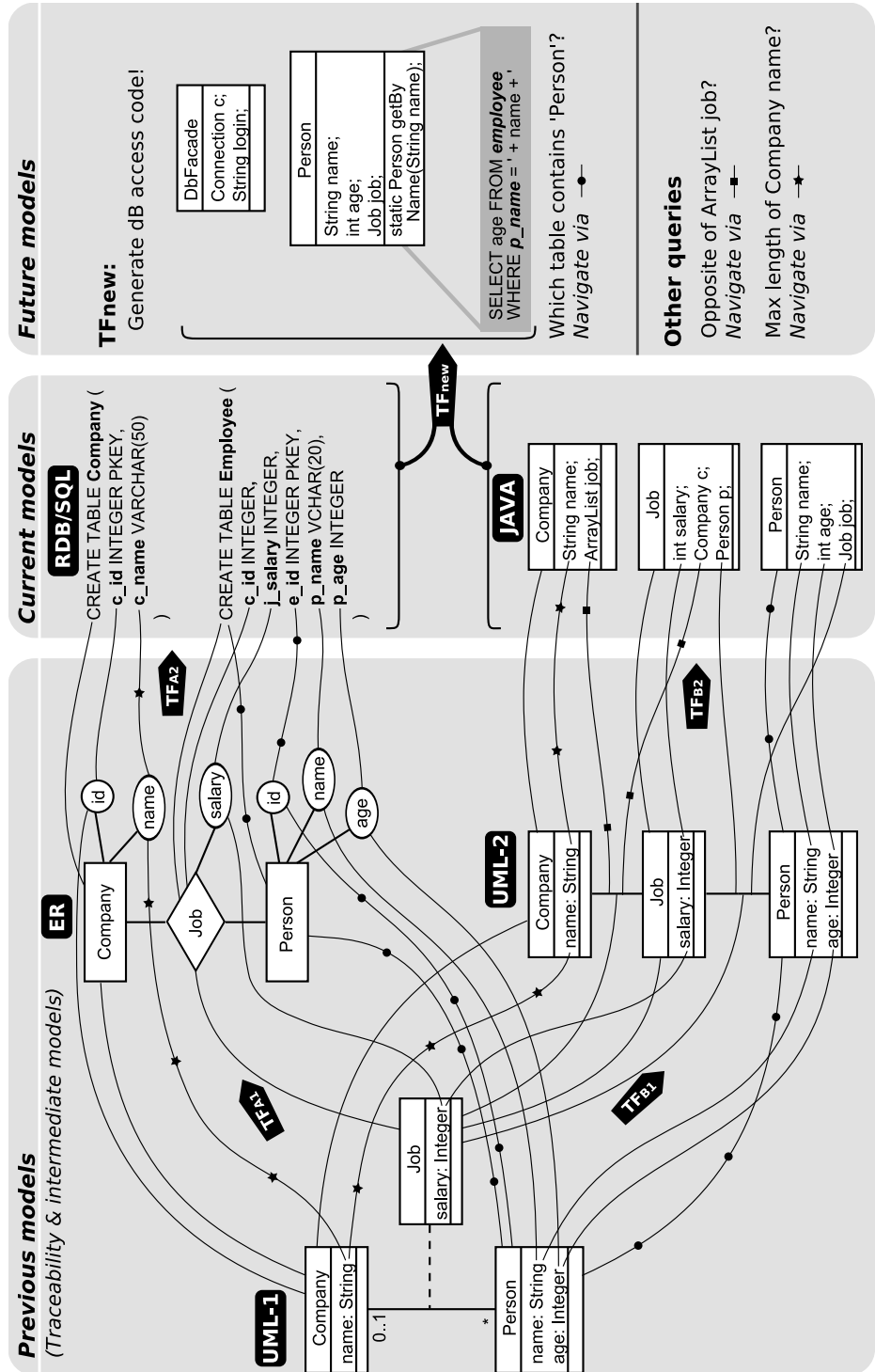


Fig. 3. Using traceability in a transformation chain to generate a relational persistence layer.

model structure to the JAVA class structure. It is easy to see that we can also use the traces to find corresponding table fields for the attributes of the *Company* and *Job* class, for example a *Job*'s *salary* attribute corresponds to *j\_salary* column in the database. Because of space restrictions we leave it up to the reader to find other relations hidden in the traceability information. Two other examples are given at the bottom rightmost corner of Figure 3.

In this and the previous subsection we have shown how we can extract valuable information from a global traceability graph that cannot be extracted from models individually but is sometimes required to perform meaningful transformations. A key observation that can be made at this point is that, although we heavily use the traceability models to derive relations, a transformation only needs to consider its actual inputs. The navigation through the traceability models can be handled transparently by a generic 'trace query' operation. As a consequence, a transformation does not have to depend explicitly on each one of the previous (traceability) models (see also Section 5).

#### 4 Producing Unambiguous Traceability Models

As allowed by Definition 1, a trace navigation can return more than one result. In many cases the required element can be selected by filtering on the element's type (see also Section 3.2). If the result of a navigation yields several elements of the same type however, the obtained information will be ambiguous. We retake the example where an accessor method is generated for each class attribute and we now include generation of mutator methods (i.e. a setters) in a separate transformation (see Figure 4). If we start navigation from attribute *MyAttr*, we get a collection of two operations as result: both mutator and accessor (see dotted arrows). Since they are of the same type (*Operation*), it is impossible to filter out the desired result.

To solve this issue we need to add additional semantics to the traces that allow to distinguish different trace paths. In case of the example we need to be able to find out which trace path leads to the accessor and which leads to the mutator. Since we use a generic traceability metamodel, an appropriate way to add these semantics is by applying the trace tagging approach (explained in 2.1). This way we do not limit the possible semantics of the traces by the traceability metamodel; these semantics are very dependent on the type of transformation and it seems quite impossible to capture all possibilities a priori as in a pure metamodel approach (see 2.1).

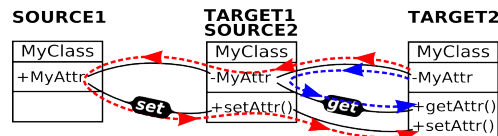


Fig. 4. Navigating tagged trace links.

In figure 4 we apply two consecutive transformations: first we add mutators (TARGET1) and subsequently we add accessors (TARGET2) and tag the traces appropriately

with ‘set’ and ‘get’. Further transformations can now unambiguously find both mutators and accessors for each attribute in model TARGET2 by navigating via the appropriate traces. As long as a ‘set’ trace is encountered on the navigation path we get the mutator; when a ‘get’ tag is encountered, we get the accessor. We refer to a navigation that takes tagged traces into account as a qualified navigation. The following definition uses the same assumptions as Definition 1.

**Definition 2.** A *qualified trace navigation* for tag  $t$  is a navigation where each path from input vertex  $v_i$  to any of the result vertices in  $V_o$  contain at least one edge (trace) tagged with  $t$ .

In case the tagged trace models, produced by previous transformations, still yield ambiguous trace navigation results it is up to the transformation assembler (who specifies a transformation composition) to decorate the traces with additional tags.

## 5 Integrating Traceability in Transformations

We have shown that automatic model transformations can generate traceability models at a very low cost and that we can navigate through traces to extract useful information. In this section we identify three transformation areas that need to be extended in order to fully implement our approach. We also propose early ideas to extend these areas.

**Production of Traces** It is not very hard to produce traceability models as part of model transformations. Some transformation languages even offer dedicated support for this purpose (for an overview, see [2]). Nevertheless, we believe that it would be useful to see how this can be improved, keeping the trace tagging metamodel in mind. Adding dedicated trace tagging syntax to transformation languages could be very useful, for example allowing annotation of each transformation target element with a tag that is transferred to a corresponding trace at execution time (see Listing 1.1).

```
rule Attr2Set
  source (Attribute aIn)
  target (Operation oOut) tag 'set'
```

**Listing 1.1.** Example syntax for trace tagging (in pseudo code).

**Traceability Queries** In order for transformations to access traceability information (through the global traceability graph), they will need to include both traceability models (edges) and past regular models (vertices) as additional inputs. This has very negative effects on various aspects of the transformation. First of all reusability goes down since the transformation is directly dependent on all the intermediate results produced by previous transformations. Secondly, the specification of the transformation becomes cluttered with many additional models that are not used directly in the transformation but are solely used to derive relations between the real input models. Finally, trace navigation needs to be encoded in the transformation itself, cluttering its implementation.

However, as briefly mentioned in Section 3.2, a transformation does not have to depend directly on all the previous models if we factor out the (qualified) trace navigation operation. We propose to introduce the *traceVia(tag: Set(String)): Set(oclAny)* operation in OCL as the single access point to the traceability information. In Listing 1.2 we use this operation to find out if an operation is a mutator. This approach would eliminate most of the disadvantages described in the previous paragraph.

```

rule RenameMutators
  source (Operation in)
    when in.traceVia('set')->select(oclIsTypeOf(Property))->
      notEmpty()
  target (Operation out)
    out.name = 'my' + in.name

```

**Listing 1.2.** Example syntax for trace navigation (in pseudo code).

**Declaration of Traceability Usage** Each transformation needs to have a clear specification so that it can easily be reused and composed in a transformation chain. In the first place, this comes down to a specification of in and output models. But, if transformations also depend on traceability information from previous transformations, the specification also has to include that information. We think that this can be accomplished by specifying required and provided trace tags. Each transformation can define its own tags independently of other transformations, improving reusability. Whenever the transformation assembler connects transformations it might then be necessary to do a semantic mapping between required and provided tags. More research is needed to find a suitable set of mapping strategies.

## 6 Related Work

In previous work [6] we have proposed a technique that uses UML Profiles to encode intra-model traceability relations directly in a UML model. The approach taken in this paper extends this work by externalizing traces, allowing for inter-model relations and metamodel independence.

Most current transformation languages [7, 3, 8] build an internal traceability model that can be interrogated at execution time, for example, to check if a target element was already created for a given source element. This approach is specific to each transformation language and sometimes to the individual transformation specification. The language determines the traceability metamodel and the transformation specification determines the label of the traces (in case of QVT/Relational the traceability metamodel is deduced from the transformation specification). The approach taken only provides access to the traces produced within the scope of the current transformation. The aim of our approach is to open up this information to other subtransformations.

Marvie describes a transformation composition framework [9] that allows manual creation of *linkings* (traces). These linkings can then be accessed by subsequent transformation, although this is limited to searching specific traces by name, introducing tight coupling between subtransformations. Our proposal allows a more loose coupling

by matching tags within a trace navigation and is able to deduce information that spans more than a single trace.

In [10], a traceability framework for Kermeta is discussed. This framework supports the creation of traces throughout a transformation chain. However, the authors do not discuss how this information can be exploited in subsequent transformations.

## 7 Conclusions & Future Work

Currently, most model transformations do not take the results of previous transformations into account. These intermediate results are a collection of regular and traceability models; the latter can be produced at very low cost in an MDD setting.

In this paper, we considered intermediate models as part of a *global traceability graph* and showed that this graph can be used to derive useful relations between (inter) an within (intra) input models of subsequent transformations. These relations cannot be derived from individual input models but are sometimes required to specify meaningful transformations. We defined the notion of (*qualified*) *trace navigation* as a meta-model independent way to search for such relations. We introduced semantic annotations (*tags*) on the traceability models to avoid ambiguities during trace navigation.

Finally, we discussed three areas in transformation techniques that need to be extended in order to enable the use of traceability in transformation development. These are integration of trace creation (1) and trace navigation/querying (2) in transformation languages and (3) declaring traceability usage in a transformation specification. We are currently extending ATL [3] with the *traceVia* operation that we proposed as a solution for (2). We will use this extended version of ATL to further validate our approach.

## References

1. Gills, M.: Survey of traceability models in it projects. In: ECMDA-TW Workshop. (2005)
2. Czarniecki, K., Helsen, S.: Classification of model transformation approaches. In: OOPSLA 2003 WS on Generative Techniques in the context of Model Driven Architecture. (2003)
3. Jouault, F., Kurtev, I.: Transforming models with atl. In: Proceedings of the Model Transformations in Practice Workshop at MoDELS 2005, Montego Bay, Jamaica (2005)
4. Ceri, S., Comai, S., Damiani, E., Fraternali, P., Paraboschi, S., Tanca, L.: XML-GL: A graphical language for querying and restructuring XML documents. In: Sistemi Evoluti per Basi di Dati. (1999) 151–165
5. Thalheim, B., Thalheim, B.: Entity-Relationship Modeling: Foundations of Database Technology. Springer-Verlag New York, Inc., Secaucus, NJ, USA (2000)
6. Vanhooff, B., Berbers, Y.: Supporting modular transformation units with precise transformation traceability metadata. In: ECMDA-TW Workshop, SINTEF, (2005) 15–27
7. Object Management Group: Qvt-merge group submission for mof 2.0 query/view/transformation. Misc (2005)
8. Lawley, M., Steel, J.: Practical declarative model transformation with tefkat. In: MoDELS Satellite Events. (2005) 139–150
9. Marvie, R.: A transformation composition framework for model driven engineering. Technical Report LIFL-2004-10, LIFL (2004)
10. Falleri, J.R., Huchard, M., Nebut, C.: Towards a traceability framework for model transformations in kermeta. In: ECMDA-TW Workshop. (2006)