Variable-Based Analysis for Traceability in Models Transformation

Omar Martínez Grassi, Claudia Pons

LIFIA, Facultad de Informática, UNLP, CAETI - Universidad Abierta Interamericana (UAI)

omartinez@rionegro.com.ar, cpons@lifia.info.unlp.edu.ar

Abstract. Model-driven development (MDD) is a software engineering approach consisting of models and their transformations. MDD gives the basic principles to visualize a software system as a set of models that are repeatedly refined until reaching a model with enough details to implement. Model-driven architecture (MDA) is the MDD view of Object Management Group. MDA main goal is to separate the system functional specification from the implementation specification on an given platform. Traceability, as a desired feature of transformations, has a major role within the paradigm since it allows the possibility to evaluate the impact at advanced stages of changes in requirement specification that were elicited early, and keeping consistency between models that guide the development, among other benefits. This paper proposes a mechanism to get traceability information from a transformation definition written in QVT language using a trace inference strategy defined ad hoc. This process is fully automated and does not depend on the execution of the transformation.

1 Introduction

The Model Driven Architecture (MDA) is a software development framework defined by the Object Management Group. The main concept to MDA is the importance of models in the software development process, and their transformations [8]. Within MDA, the software development process is driven by the activity of modeling the software system. MDA proposes a development cycle based on the transformation of a high-level model into another, with a lower level of abstraction, which eventually will become source code.

The first model MDA defines is a model of a high level of abstraction that is independent of any implementation technology. This is called Platform Independent Model or PIM. In the next step, the PIM is transformed into one or more Platform Specific Models (PSM). A PSM specifies a system in terms of the implementation constructs available in one specific implementation technology. The final step in the development process is the transformation of each PSM into code. The MDA defines the PIM, PSM and code, and also defines how these relate to each other. A PIM should be created, then transformed into one or more PSMs, which then are transformed into code. A model transformation is a process described by a definition consisting of rules, which specify how a source model element is mapped into another target model element.

The MDA process may look like traditional development. However, there is a crucial difference: traditionally, the transformation from model to model, or from model to code, is done mainly by hand. In contrast, MDA transformations are always executed by tools. Many tools are able to transform a PSM to code, there is nothing new. What is new in MDA is that the transformation from PIM to PSM is automated as well. In particular, we are interested in the study of the property of traceability in model transformations.

Some years ago, the OMG adopted QVT (Query/View/Transformation) language as a standard of model transformation. QVT is a hybrid declarative/imperative language [9], which integrates the standard OCL 2.0 and extends its imperative version, defining three specific domain languages (DSL) called Relations, Core (both declarative) and Operational Mappings (imperative). Unfortunately, there are not many tools that implement QVT languages. We can find mediniQVT [1] as a QVT Relation implementation, SmartQVT [11] (QVT Operational Mappings) and OptimalJ (QVT Core), for example. In this context, the Eclipse Modeling Framework (EMF) Project [3] provides a modeling environment and code generation for application development based on models that can be specified using a subset of the Java language (known as Java Annotated), XML documents or modeling tools such as Rational Rose ™. The project includes Ecore, an implementation of Meta Object Facility Standard (MOF) [9], a fundamental tool for model representation.

This paper makes a proposal for traceability support in model transformation using QVT code analysis, which allows the inference of traces between the source and target models from the specification of the transformation, systematically, without requiring additional code nor intervention from the developer.

2 Traceability in model transformations

2.1 The traceability concept

The IEEE Standard Glossary of Software Engineering Terminology [10] defines traceability as follows:

1. The degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor-successor or master-subordinate relationship to one another; for example, the degree to which the requirements and design of a given software component match;
2. The degree to which each element in a software development product establishes its reason for existing; for example, the degree to which each element in a bubble chart references the requirement that it satisfies.

This early definition is strongly influenced by the originators of traceability, i.e. requirements management community. However, it is possible to find a much broader one, more useful for the purposes of model-driven development. In [2], Aizenbud defines traceability as "any relationship that exists between artifacts involved in the software engineering life cycle". In addition, as the author explains, this definition includes, but it is not limited to the following:

- Explicit links or mappings that are generated as a result of transformations, both forward (e.g., code generation) and backward (e.g., reverse engineering).
- Links that are computed based on existing information (e.g., code dependency analysis).
- Statistically inferred links, which are links that are computed based on history provided by change management systems on items that were changed together as a result of one change request.

So traceability is achieved by defining and maintaining relationships between artifacts involved in the software-engineering life cycle during system development.

2.2 Related work

The automatic generation of traceability information has been the subject of several research papers. One of the first studies of traceability in model transformations can be found in [7]. It is based on traces generation through a loosely coupled process, without altering the definition of model transformations in the context of language ATL (ATLAS Transformation Language), a model transformation language [6] stood as a candidate in the RFP (Request For Proposal) of QVT launched by the Object Management Group (OMG) [5].

A more complex approach can be found in [4]. In this study, Grammel et al. proposes a generic traceability framework for augmenting arbitrary model transformation approaches with a traceability mechanism. This generic traceability framework is based on a domain-specific language for traceability (Trace-DSL), presenting the formalization on integration conditions needed for implementing traceability. Essentially, this language agnostic Trace-DSL provides a unified traceability metamodel, yet accounts for an adequate expressiveness of traceability data needed for traceability-specific scenarios. To achieve this dual nature, the Trace-DSL is featured with an extensibility mechanism based on facets. The work covers a wide range of traceability aspects at the model-driven development paradigm, and proposes a generic solution for different model transformation languages.

In Section 5 we will discuss further details of both implementations, and will compare them with the scheme presented in this work.

3 QVT Relations

To understand the proposal we need to review some core concepts of the language in which the analyzed model transformations are written. In this section, we will make a brief review of the language definition and its generalities.

3.1 Transformations and model types

In the relations language, a transformation between candidate models is specified as a set of relations that must hold for the transformation to be successful [9]. A candidate model is any model that conforms to a model type, which is a specification of what kind of model elements any conforming model can have, similar to a variable type specifying what kind of values a conforming variable can have in a program. Candidate models are named, and the types of elements they can contain are restricted to those within a set of referenced packages. An example is:

transformation umlRdbms(uml:SimpleUML, rdbms:SimpleRDBMS)

In this declaration named "umlRdbms" there are two typed candidate models: "uml" and "rdbms". The model named "uml" declares the SimpleUML package as its metamodel, and the "rdbms" model declares the SimpleRDBMS package as its metamodel. A transformation can be invoked either to check two models for consistency or to modify one model to enforce consistency.

3.2 Relations and Domains

Relations in a transformation declare constraints that must be satisfied by the elements of the candidate models. A relation, defined by two or more domains and a pair of *when* and *where* predicates, specifies a relationship that must hold between the elements of the candidate models.

```
top relation PackageToSchema {
  domain uml p:Package {
    name = pn
  };
  domain rdbms s:Schema {
    name = pn
  };
}
```

Fig. 1. Relations and domains example

A domain is a distinguished typed variable that can be matched in a model of a given model type. A domain has a pattern, which can be viewed as a graph of object nodes, their properties and association links originating from an instance of the domain's type. Alternatively a pattern can be viewed as a set of variables, and a set of constraints that model elements bound to those variables must satisfy to qualify as a valid binding of the pattern. A domain pattern can be considered a template for objects and their properties that must be located, modified, or created in a candidate model to satisfy the relation.

```
relation ClassToTable {
  domain uml c:Class {
    namespace = p:Package {},
    kind = 'Persistent'
    name = cn
  };
  domain rdbms t:Table {
    schema = s:Schema {},
    name = cn,
    column = cl:Column {
      name = cn + '_tid',
      type = 'NUMBER'
    },
    primaryKey = k:PrimaryKey {
      name = cn + '_pk',
      column = cl
    }
  };
  when {
    PackageToSchema(p,s);
  }
  where {
    AttributeToColumn(c,t);
  }
}
```

Fig. 2. *When* and *Where* clauses example

In the example at Figure 1 two domains are declared that will match elements in the “uml” and “rdbms” models respectively. Each domain specifies a simple pattern, a package with a name, and a schema with a name, both the “name” properties being bound to the same variable “pn” implying that they should have the same value.

3.3 When and Where clauses

A relation also can be constrained by two sets of predicates, a *when* clause and a *where* clause, as shown in the example relation ClassToTable (Figure 2). The *when* clause specifies the conditions under which the relationship needs to hold, so the relation ClassToTable needs to hold only when the PackageToSchema relation holds between the package containing the class and the schema containing the table. The *where* clause specifies the condition that must be satisfied by all model elements participating in the relation, and it may constrain any of the variables in the relation and its domains. Hence, whenever the ClassToTable relation holds, the relation AttributeToColumn must also hold.

The *when* and *where* clauses may contain any arbitrary OCL expressions in addition to the relation invocation expressions. Relation invocations allow complex relations to be composed from simpler relations.

3.4 Top-level Relations

A transformation contains two kinds of relations: top-level and non-top-level. The execution of a transformation requires that all its top-level relations hold, whereas non-top-level relations are required to hold only when they are invoked directly or transitively from the where clause of another relation.

```
transformation umlRdbms(uml:SimpleUML, rdbms:SimpleRDBMS) {
  top relation PackageToSchema() {...}
  top relation ClassToTable {...}
  relation AttributeToColumn {...}
  ...
}
```

Fig. 3. Top-level and non-top-level relations in QVT

A top-level relation has the keyword top to distinguish it syntactically. In the example at Figure 3, PackageToSchema and ClassToTable are top level relations, whereas AttributeToColumn is a non-top-level relation.

3.5 Check and enforce

Whether or not the relationship may be enforced is determined by the target domain, which may be marked as checkonly or enforced. When a transformation is enforced in the direction of a checkonly domain, it is simply checked to see if there is a valid match in the relevant model that satisfies the relationship. When a transformation executes in the direction of the model of an enforced domain, if checking fails, the target model is modified so as to satisfy the relationship, i.e., a check-before-enforce semantics. In the example below (Figure 4), the domain for the “uml” model is marked checkonly and the domain for the “rdbms” model is marked enforce.

```
top relation PackageToSchema {
  checkonly domain uml p:Package {
    name = pn
  };
  enforce domain rdbms s:Schema {
    name = pn
```

```
        };
      }
```

Fig. 4. Relations and domains example

If we are executing in the direction of “uml” and there is a schema in “rdbms” for which there is no corresponding package with the same name in “uml”, it is simply reported as an inconsistency. Then a package is not created because the “uml” model is not enforced, it is only checked.

However, if we are executing the transformation umlRdbms in the direction of “rdbms”, then for each package in the “uml” model the relation first checks if there is a schema with the same name in the “rdbms” model, and if there is not, a new schema is created in that model with the given name. To consider a variation of the above scenario, if we execute in the direction of “rdbms” and there is not a corresponding package with the same name in “uml”, then that schema will be deleted from the “rdbms” model, thus enforcing consistency in the enforce domain. These rules apply depending on the target domain only. In this execution scenario, schema deletion will be the outcome even if the “uml” domain is marked as enforced, because the transformation is being executed in the direction of “rdbms”, and object creation, modification, and deletion can only take place in the target model for the current execution.

4 Variables-based analysis

The present study addresses the problem of obtaining traceability information automatically, i.e. without having to depend on someone to specify how target model elements are generated from a source model or the execution of a transformation. Unlike other similar proposals, this study suggests that given the syntactic-grammatical features of QVT language[1], it is possible to infer some kind of traces by analyzing the source code. This analysis is the recognition of certain structures within the specification of a model transformation written in QVT Relations language, which can be used to discover traces between source and target model elements.

We have identified four types of traces that can be recognized with this approach:

- Simple trace: Specifies how an element from source model maps to an element in the target model (one-to-one relationship).
- Multitrace: This specifies how multiple element maps to a simple target model element (many-to-one relationship).
- Conditional trace: This kind of trace represents those potential traces that can not be confirmed because they respond to a conditional statement (e.g. if-then-else structure) within the QVT code and therefore depend on the transformation.

[1] OMG standard model transformation language

- Constant trace: This type of trace models situations where a target element assumes a constant value in a transformation specification.

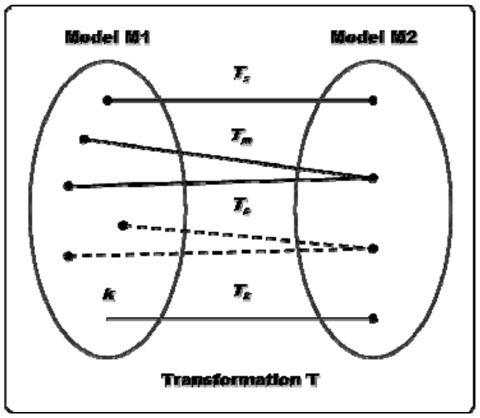


Fig. 5. Different kind of traces detected with our approach

Figure 5 shows the four kind of traces that can be detected with our mechanism. Let M1 and M2 be two different models, and T a QVT transformation that defines a conversion from M1 to M2, then a simple trace T_s specifies the mapping of a single element from source model, M1, on an element of target model M2, in a transformation T. A multiple trace T_m represents a many-to-one relationship between multiple elements from source model and a single element from destination model. A conditional trace T_c specifies two potential traces between a couple disjoint sets of elements from source model and a target model element. Finally, a constant trace T_k models the assignment of a constant value on a target model element within a transformation definition.

This section describes the features that allow the inference of all these kind of traces.

4.1 Trace inference analysis

As we point out previously, our work is based on the hypothesis that it is possible to infer traces directly from QVT code by implementing an algorithm for identification of certain grammatical or lexical structures, or patterns, in the specification of a model transformation. We will discuss now the patterns that allow traces derivation, illustrating each case with QVT source fragments in which they are present.

Case No.1: Trace inference using an auxiliar variable.

When a top-level rule, or a non-top-level rule invoked from a statement of a *when*, or *where*, clause of a top-level rule, assigns a value to a target model element defined in the scope of an enforce domain, by using a variable previously used on a source model element defined similarly in a checkonly domain, then we say that the source model element will map directly to the target model element. If we take the QVT code fragment in Figure 6a, we see that the variable called *pn* allows to infer a trace between *umlName* and *rdbmsName*, attributes of entities *UmlPackage* and *RdbmsSchema* respectively.

```
top relation PackageToSchema {
  pn : String;
  checkonly domain uml p:SimpleUML::UmlPackage
  {
    umlName = pn
  };
  enforce domain rdbms s:SimpleRDBMS::RdbmsSchema
  {
    rdbmsName = pn
  };
}
```

a. Trace inference using an auxiliar variable

```
relation ClassToPkey {
  cn : String;
  checkonly domain uml c:SimpleUML::UmlClass
  {
    umlName = cn
  };
  enforce domain rdbms k:SimpleRDBMS::RdbmsKey
  {
    rdbmsName = cn + '_pk'
  };
}
```

b. Trace inference using a function of an auxiliar variable

Fig. 6. Trace inference Cases No.1 and No.2

Case No.2: Trace inference using an expression in terms of an auxiliary variable.

This case is a generalization of the described above, the difference is based on the target model element, defined in the scope of the enforce domain, will be a function of the variable used for the same purpose on the source model element (Figure 6b). As can be seen, the expression describing the value that the attribute named rdbmsName will take after transformation is given by a function *F* of the variable *cn*, defined as *F(cn) = cn + '_pk'*, where the operator '+' represents string concatenation. In this case, we can infer that all attribute *rdbmsName* of a *RdbmsKey* entity of a *SimpleRDBMS* model will be equal to the entity *UmlClass* attribute, called *umlName*, from *SimpleUML* model, concatenated with the suffix '_pk' or, equivalently, that *rdbmsName = umlName + '_pk'*.

```
top relation ClassToTable {
  ...
  enforce domain rdbms t : SimpleRDBMS::RdbmsTable {
    rdbmsSchema = s : SimpleRDBMS::RdbmsSchema {},
    rdbmsName = cn,
    rdbmsColumn = cl : SimpleRDBMS::RdbmsColumn {
      rdbmsName =cn + '_tid',
      rdbmsType = 'NUMBER'
    },
    rdbmsKey = k : SimpleRDBMS::RdbmsKey {
      rdbmsColumn = cl : SimpleRDBMS::RdbmsColumn{}
    }
  };
  ...
}
```

Fig. 7. Trace inference using a constant (Case No.3)

Case No.3: Trace inference using a constant.

It is defined for those cases in which a target model element defined in the scope of the enforce domain of a relation is initialized to a constant value. Taking the example of the code presented in Figure 7, we see that the attribute *rdbmsType* of any entity *RdbmsColumn* be equal to the constant 'NUMBER', regardless of the values of the associated source model elements.

Case No.4: Trace inference using an auxiliary variable defined as a function in a Where clause.

This case is analogous to the first kind of trace, described in Case No.1. The difference is that the auxiliary variable is defined as a function of other variables in a statement in the *Where* clause of the relation (see Figure 8). In this case we infer both that the name (attribute *rdbmsName*) of a column (entity *RdbmsColumn*) within a foreign key (entity *RdbmsForeignKey*) will be the concatenation of the source class (entity *umlSource*) name (attribute *umlName*) with the symbol '_' to the name (attribute *umlName*) of the association (entity *UmlAssociation*), concatenated in turn with the symbol '_' to the name (attribute *umlName*) of the target class (entity *umlDestination*) + '_tid', and that the name (attribute *rdbmsName*) of the foreign key (entity *rdbmsForeignKey*) is analogous to the former, without suffix '_tid'.

```
top relation AssocToFKey {
  an, scn, dcn, fkn, fcn : String;
  checkonly AssocToFKeydomain uml a:SimpleUML::UmlAssociation {
    umlNamespace = p : SimpleUML::UmlPackage {},
    umlName = an,
    umlSource = sc : SimpleUML::UmlClass {
      umlKind = 'Persistent',
      umlName = scn
    },
    umlDestination = dc : SimpleUML::UmlClass {
      umlKind = 'Persistent',
      umlName = dcn
    }
  };
  enforce domain rdbms fk : SimpleRDBMS::RdbmsForeignKey {
    rdbmsName = fkn,
    rdbmsOwner = srcTbl : SimpleRDBMS::RdbmsTable {
      rdbmsSchema = s : SimpleRDBMS::RdbmsSchema {}
    },
    rdbmsColumn = fc : SimpleRDBMS::RdbmsColumn {
      rdbmsName = fcn,
      rdbmsType = 'NUMBER',
      rdbmsOwner = srcTbl
    },
    rdbmsRefersTo = pKey : SimpleRDBMS::RdbmsKey {
      rdbmsOwner = destTbl : SimpleRDBMS::RdbmsTable {}
    }
  };
  when {
    ClassToPkey(dc, pKey);
    PackageToSchema(p, s);
    ClassToTable(sc, srcTbl);
    ClassToTable(dc, destTbl);
  }
  where {
    fkn = scn + '_' + an + '_' + dcn;
    fcn = fkn + '_tid';
  }
}
```

Fig. 8. An auxiliar variable and expression within a *Where* clause (Case No.4)

Case No.5: Trace inference by a conditional If-Then-Else statement.

The following case is defined for those situations where a mandatory rule (top-level), or a non-top-level rule invoked from a statement of a *when*, or *where*, clause of a top-level rule, assigns a value to a target model element defined in the scope of an enforce domain by using a conditional statement If-Then-Else, whose expression within “then” clause (or “else” clause) includes a variable previously used on a source model element similarly defined in the scope of a checkonly domain. In this case, we say that the source model element will map conditionally, or partially, in the target model element.

```
relation PrimitiveAttributeToColumn {
  an, pn, cn, sqltype : String;
  checkonly domain uml c : SimpleUML::UmlClass {
    umlAttribute = a : SimpleUML::UmlAttribute {
      umlName = an,                                      (1)
      umlType = p :
      SimpleUML::UmlPrimitiveType {
        umlName = pn
      }
    }
  };
  enforce domain rdbms t : SimpleRDBMS::RdbmsTable {
    rdbmsColumn = cl : SimpleRDBMS::RdbmsColumn {
      rdbmsName = cn,                                    (2)
      rdbmsType = sqltype                                (6)
    }
  };
  primitive domain prefix : String;
  where {
    cn = if ( prefix = _ ) then                          (3)
      ( an )                                             (4)
    else
      ( prefix + '_' + an )                              (5)
    endif;
    sqltype = PrimitiveTypeToSqlType(pn);                (7)
  }
}
```

Fig. 9. Cases No.5 y No.6 example

Figure 9 shows a conditional trace example at PrimitiveAttributeToColumn relation. Attribute *rdbmsName* from *RdbmsColumn* is assigned to a variable *cn* (2), which is initialized within *Where* clause with a conditional statement (3) depending on the value of *prefix* primitive domain. This way, replacing value of variable *an*, we can infer the following trace:

$$an \mid prefix + \text{‘\_’} + an \rightarrow rdbmsName$$

This means that for any couple *(UmlClass, RdbmsTable)* that satisfies the PrimitiveAttributeToColumn relation, then the name of each column of resultant table (*rdbmsName* attribute) will be equal to the name of the class attribute (*umlName* attribute) from origin (4), eventually preceded or not by string character *prefix* (5).

Case No.6: Trace inference by a query.

The sixth case is defined for those situations where a mandatory rule (top-level), or a non-top-level rule invoked from a statement of a *when*, or *where*, clause of a top-level rule, assigns a value to a *PrimitiveTypeToSqlType()* target model element defined in the scope of an enforce domain by using a query, directly from a domain expression or by using an assigned variable within *where* clause of the relation, as well. In the example at Figure 9, the algorithm allow us to infer simple trace:

PrimitiveTypeToSqlType(umlName::UmlPrimitiveType)→rdbmsType::RdbmsColumn

This means that for any couple *(UmlClass, RdbmsTable)* that satisfies the *PrimitiveAttributeToColumn* relation, the data type of any column (*rdbmsType* attribute) of the table (6) will be given by the query *PrimitiveTypeToSqlType()*, a function that take a class attribute data type and returns the SQL equivalent data type (7).

4.2 Advantages and disadvantages of variable-based analysis

The technique presented here is fully automatic, i.e. no stage of the analysis process requires the intervention of a human being to operate. As such, it colaborates with the productivity of enginners by removing effort and possible errors. Unlike some implementations, such as mediniQVT [1], traceability information is generated at model level, not instances, so it allows to determine not only the mapping of one element into another, but expression or form of this transformation, regardless of the source model instance to be transformed or the corresponding target model instance. This, in turn, allows to verify and eventually force the consistency and integrity of the relationship between the two models, which can be especially helpful when the target model is modified unilaterally, and not as a result of changes in the source model then processed by the transformation engine, which would be the natural flow of the modification process.

Another advantage observed is that obtained traces do not depend on transformation process, but only its definition. Consequently, this can help the developer as a debugging tool in the depuration of the transformation specification, providing indications of the results to be obtained after the execution. This independence, in turn, provides flexibility and facilitates maintaining traceability information since it can be stored in a repository, or generated ad hoc without polluting neither models nor transformation specification.

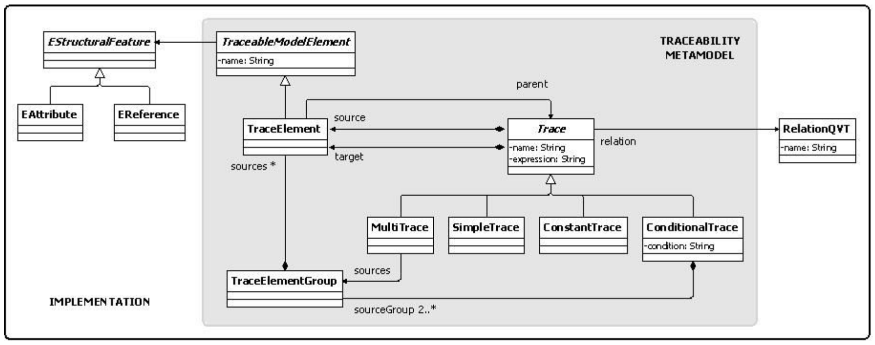


Fig. 10. Traceability metamodel implemented by QVTrace

The main disadvantage of the proposal is that analysis of the specification of a transformation and source and target models, should be performed twice: for the analysis based on variables (the tool that implements it), and at the time of execution of the processing by the engine that implements QVT. Moreover, the generated traces are not necessarily the only existing. This paper does not attempt to demonstrate so.

4.3 QVTrace

The described approach has been implemented in a tool called QVTrace. It is an Eclipse plugin, which despite being a prototype allowed us to implement this technique and check its applicability in the context of MDD (Model-Driven Development). QVTrace has been developed with a vision to be a complementary tool to other available at model-driven development paradigm. Being an Eclipse plugin, offers versatility and enhances interoperability with other related programs. Its inputs are the definition of the transformation (QVT code) and source and destination models in Ecore format, standard representation of models in the EMF [3], which is the framework for model-driven development of Eclipse, and the generated output is a collection of traces defined in an ad-hoc metamodel. This implementation includes all six trace inference cases described in this paper.

Traceability Metamodel.

To maintaining traceability information a metamodel based on simplicity was developed, tailored to the needs of the problem addressed (Figure 10). It consists of a class named Trace, which maintains all the information associated with a trace:

- The name of the trace, an identifier formed by the name of the model elements that compose it (sourceElement2targetElement).
- Source and target model elements that conforms the trace (references *source* and *target* in the diagram of Figure 10).

- The expression of the trace (*expression* attribute), a character string that represents the traceability relationship between source model element and destination model element of the trace.
- The relation in QVT code where the trace was found (reference *relation*).

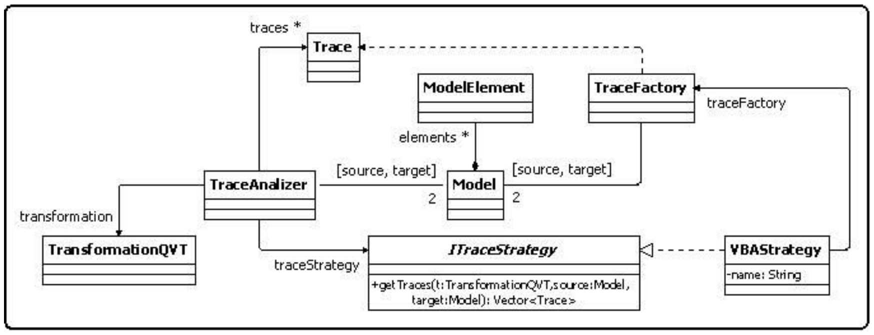


Fig. 11. Trace inference support in QVTrace

The elements related by a trace are TraceElement type, which are in turn TraceableModelElement subtypes, an abstract class that determines what kind of model element can be included in a trace. This design is closely related to the representation of models used. In the specific case of QVTrace, Ecore model representation was used, the EMF model representation standard. Every TraceableModelElement object contains a reference to an EStructuralFeature object, so a trace will only relate EAttribute or EReference objects from source and target models, as we will see later.

In contrast to the metamodels proposed in similar works, the developed traceability model presented here represents traces as an univocal relationship between a source and a destination model elements, while in most cases, this relationship is generalized as a many-to-many relationship. The proposal does provide a possible trace of *n* source model elements to one target model element, which is typified by MultiTrace class, subclass of Trace. This approach, which could be considered a limitation by design, actually responds to a virtue. The trace inference algorithm works at the minimum traceable element level in the context of model representation chosen, Ecore, and by the features of variable-based analysis, if one or more source model elements generate multiple destination model elements, then multiple traces MultiTrace or Trace will generate, as appropriate.

The second difference with most proposed metamodels in related works has to do with the semantics of the trace. One of the attributes of the Trace class, called expression, stores the statement that reveals the meaning of transformation, i.e. how a source model element is transformed into a given target model element. As an example, let A2B be a transformation where element x from model A is converted into an element y from model B, so we have a trace $x \rightarrow y$ in which expression will be the statement $y=x$, adding the meaning to relationship.

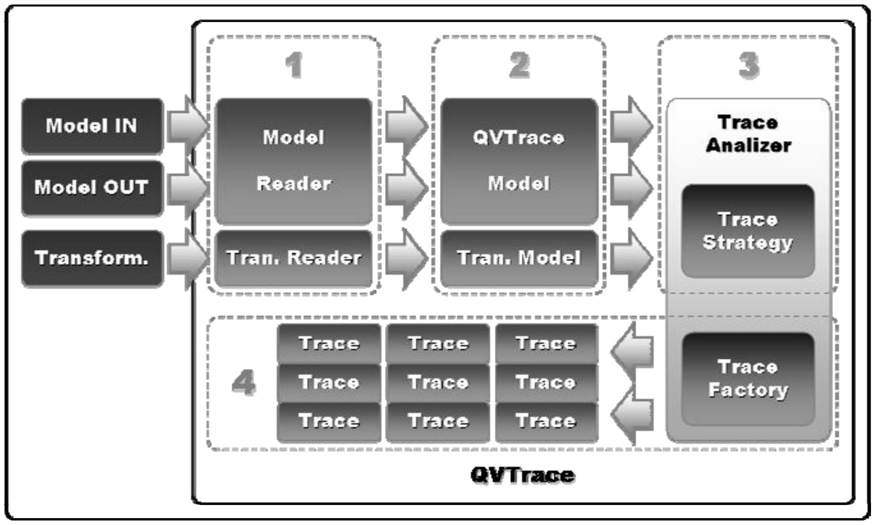


Fig. 12. General view of QVTrace

Trace inference support.

Trace inference in QVtrace is supported through the scheme proposed in Figure 11. It consists of a component called TraceAnalizer, which using the source and target models (object Model), the definition of a QVT transformation (object TransformationQVT), and a traceability strategy (components that implement the interface ITraceStrategy) infers and generates the corresponding traces. Tracing strategy is essentially the mechanism by which corresponding traces are obtained. The design is intended for this component can be easily extended or replaced by another one that implements the interface method *getTraces()*, which as shown in the picture receives a QVT transformation, a pair of source and target models, and returns results in a collection (Vector) of Trace objects. The responsibility for creating trace objects is in charge of TraceFactory component, which is the Trace objects maker (see arrow with a dotted line in the diagram). Any traceability strategy implemented must use this object factory for creating them.

Solution design.

In summary, the proposed QVTrace workflow begins processing the input data, and finish with obtained traces. This process can be divided into four phases:

1. Reading and parsing the source and target models, presented in Ecore format, and the QVT transformation specification, from files.
2. Creating internal representation objects for input and output models, and transformation.

3. Analyzing of transformation definition and trace inference according to TraceStrategy strategy used.
4. Creating Trace objects through the TraceFactory factory.

Figure 12 shows schematically QVTrace components. The process inputs consists of the source and target model files (in Ecore format), and QVT transformation code, which are used for trace inference. Obtaining traces is performed by a component called TraceAnalizer, in collaboration with two fundamental objects in the process: one of TraceStrategy type, which implements the strategy used for the inference of the traces, and other TraceFactory kind, which is responsible for creating traces. Thus, we decouple the creation of the trace, and therefore the knowledge of the metamodel, from the trace inference process.

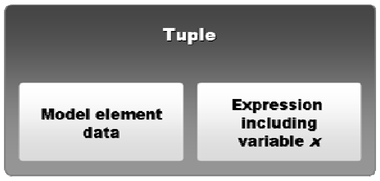


Fig. 13. Tuple date structure

4.4 Variable-based analysis implementation

Previously in this section we have presented the inference mechanism used by QVTrace. This analysis is based on the study of the use of variables within a QVT program code to infers implicit traces in the definition of the model transformation. Now we will detail how the variable-based analysis was implemented.

Overview.

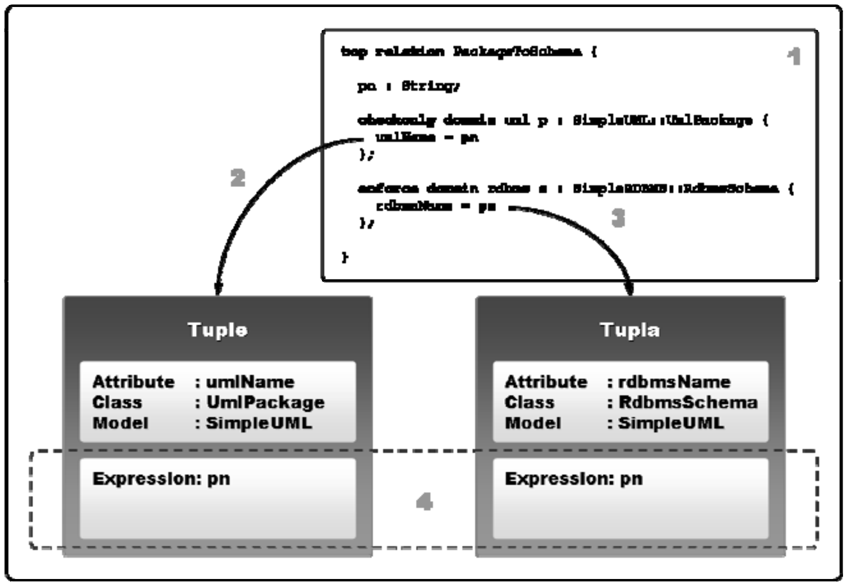
The variable-based analysis core is to determine how the variables defined in the context of a relationship or transformation rule link source model elements with elements of the target model. As explained earlier, a relationship will determine a trace, if it meets the following conditions:

1. The existence of an expression to link a source model element with a variable of the transformation rule, say x , defined in the context of a checkonly domain.
2. The existence of an expression relating a target model element to said variable x in the context of a domain enforce, belonging to the same transformation rule of the first point (1).

The expression that relates variables with model elements may take several forms, which have already explained in the trace cases that have been identified in Section 4.1.

Tuple concept.

The inference trace scheme developed is based on a data structure which was specially designed, which maintains the relationships between variables and model elements. This structure was called tuple (Figure 13), and it contains model elements information (e.g. name attribute, class to which it belongs, model which corresponds to the class, etc) and the expression which contains the variable that is related. Therefore it is possible that the expression associated with the model element contains a constant rather than a variable, as we will see later.



```
top relation PackageToSchema {

  pn : String;

  checkonly domain uml p : SimpleUML::UmlPackage {
    umlName = pn
  };

  enforce domain rdbms s : SimpleRDBMS::RdbmsSchema {
    rdbmsName = pn
  };

}
```

Fig. 14. Tuple utilization example

The trace inference strategy is then to generate the tuples present in the checkonly domain and enforce domain of every relationship, and later studying variables in common. If this condition holds, i.e. a variable that is shared by both domains exists, we will in the presence of a link (trace) from a source model element to a destination model element.

Figure 14 illustrates the use of tuples for trace inference. In the picture you can see a fragment of the definition of the rule PackageToSechema, which belongs to a transformation from an UML model into a relational one, named RDBMS. In this case, the relationship (1) specifies the conversion of UmlPackage elements (UML model) into RdbmsSchema elements (RDBMS model). As a result of variable-based analysis, two tuples are created:

1. Tuple No.1 relates source model element *umlName*, from UmlPackage class, with variable *pn* (2).

2. Tuple No.2 relates destination model element *rdbmsName*, from RdbmsSchema class, with variable *pn* too (3).

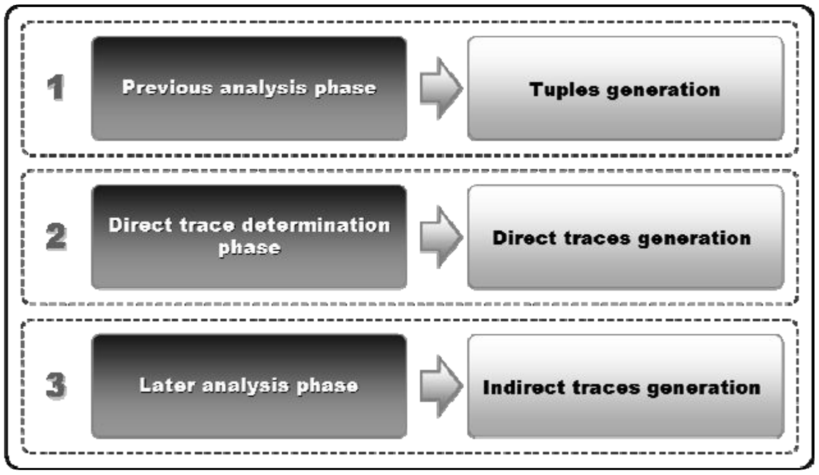


Fig. 15. Trace inference algorithm stages

Finally, the presence of variable *pn* on both tuples (4) allows to connect the two model elements, obtaining a trace. In this case, the reasoning is straightforward: if *umlName = pn* and *rdbmsName = pn* then *rdbmsName = umlName*. This way, we can be sure that any couple *(UmlPackage, RdbmsSchema)* satisfying the relation PackageToSchema verifies that the relational schema name (*rdbmsName* attribute) will be equal to the source UML package name (*umlName* attribute).

Trace inference algorithm.

The trace generation mechanism proposed is based on the analysis of the relationships variables, comprising a QVT transformation. The inference algorithm of traces consists of three stages, executed once for each variable of every relation of a QVT transformation.

- Previous analysis phase, where the tuples associated with the variable analyzed, present in checkonly and enforce domains of the relationship, are determined.
- Direct trace determination phase, where simple, constant, conditional or query traces are inferred from tuples generated in the previous step.
- Later analysis phase, where indirect traces of simple type, conditional or multiple (MultiTrace) are determined, which could not be inferred in the previous step.

Figure 15 shows the various stages of the proposed algorithm, and the results achieved by each one. Next, we will discuss them in more detail.

Previous analysis phase.

During this stage each domain of the relationship, checkonly and enforce, is analyzed for veryfing the presence of tuples associated with a given variable of the relationship. As a result of the phase, it is possible to find tuples in both domains of the relationship, or tuples in either domain, not both, depending on the use of each variable.

```
top relation PackageToSchema {
  pn : String;
  checkonly domain uml p : SimpleUML::UmlPackage {
    umlName = pn                                          (1)
  };
  enforce domain rdbms s : SimpleRDBMS::RdbmsSchema {
    rdbmsName = pn                                        (2)
  };
}
```

Fig. 16. Direct trace generation

In the former case, the presence of a tuple in the checkonly domain, and the presence of another in the enforce domain of the relationship, for the same variable, will result in a direct trace. In contrast, the presence of a tuple in an enforce domain, without corresponding to an associated tuple in the checkonly domain, or vice versa, indicating the presence of an indirect trace, can not be defined until the analysis of the post-conditions of the rule at Where clause, which is performed during the execution of the post-analysis phase, as we will see later.

Now we will present two relations corresponding to a UML2RDBMS transformation, to illustrate the above cases that arise during the previous analysis. Figure 16 shows QVT code belonging to PackageToSchema relationship, which performs the conversion of UML model entities Package into RDBMS model Schema entities. As a result of the analysis, we obtain the tuple *(umlName::UmlPackage, pn)* from the checkonly domain of the relation (1), and the tuple *(rdbmsName::RdbmsSchema, pn)* from the enforce domain (2).

The listing at Figure 17 shows a second example where no analysis results in obtaining two tuples for the same variable within domains checkonly and enforce. The source code analyzed here corresponds to the PrimitiveAttributeToColumn relationship, which specifies the mapping of a primitive attribute type element, from UML model, into a column of a table of relational model. For this case, the algorithm must carry out the analysis of the domains for each of the four defined variables. First iteration of the variable *an*, results in the tuple *(umlName::UmlAttribute, an)* from checkonly domain in (1), but there is not associated tuple in the same variable within enforce domain. In the second iteration of the analysis, on the variable *pn*, a similar situation occurs, yielding the tuple *(umlName::UmlPrimitiveType, pn)* from checkonly the domain (2) without a tuple in the enforce domain corresponding to this variable. Finally the third and fourth iteration of the previous analysis (on variables cn and sqltype respectively) allow obtaining tuples *(rdbmsName::RdbmsColumn, cn)* and *(rdbmsType::RdbmsColumn, sqltype)* from the enforce domain (3) (4), without associated tuples on the checkonly domain of relationship for the same variables.

```
relation PrimitiveAttributeToColumn {
  an, pn, cn, sqltype : String;
  checkonly domain uml c : SimpleUML::UmlClass {
    umlAttribute = a : SimpleUML::UmlAttribute {
      umlName = an,                                        (1)
      umlType = p : SimpleUML::UmlPrimitiveType {
        umlName = pn                                       (2)
      }
    }
  };
  enforce domain rdbms t : SimpleRDBMS::RdbmsTable {
    rdbmsColumn = cl : SimpleRDBMS::RdbmsColumn {
      rdbmsName = cn,                                      (3)
      rdbmsType = sqltype                                  (4)
    }
  };
  primitive domain prefix : String;
  where {
    cn = if prefix = _ then
      an                                                   (5)
    else
      prefix + '_' + an                                    (6)
    endif;
    sqltype = PrimitiveTypeToSqlType(pn);                  (7)
  }
}
```

Fig. 17. Indirect traces generation at pre-analysis phase

Table 1 summarizes the results of the execution of the previous analysis phase on the relationships analyzed. As we see, the first row shows that for the rule PackageToSchema (indicated as P2S in the table) produces two tuples related to the variable *pn*, one in the domain checkonly and one in the domain enforce the relationship. The rest of the table shows the tuples obtained at PrimitiveAttributeToColumn rule (indicated in the table as P2C), which shows that for each variable analyzed only a tuple is found in each domain.

Direct trace determination phase.

The next phase of the algorithm consists in analyzing the tuples generated during the previous step and generating direct traces. We understand for “direct trace” one that can be inferred from the tuples generated during the pre-analysis stage without any specific treatment or further study.

Relation	*Variables*	*Tuples (checkonly)*	*Tuples (enforce)*
P2S	*pn*	(umlName, pn)	(rdbmsName, pn)
P2C	*an*	(umlName, an)	-
	pn	(umlName, pn)	-
	cn	-	(rdbmsName, cn)
	sqltype	-	(rdbmsType, sqltype)

Table 1. Obtained results during the pre-analysis for each relationship

During this stage are tuples generated in the previous step are analized and it is checked whether, for a given variable, there is a tuple in both domains of the relationship, checkonly and enforce, whose expression containing them. Any trace inferred, in this case, will be a direct trace. The input data of the phase will be the list of tuples generated during the previous analysis from both domains, while the output will be direct traces of simple, conditional or constant type, which can be inferred from the tuples.

Following our previous example, we can see that relationship PackageToSchema execution of this phase succeeds obtaining the trace as follow:

$$umlName::UmlPackage \rightarrow rdbmsName::RdbmsSchema$$

inferred from collected tuples in previous phase for variable pn. In contrast, the execution of the stage for the PrimitiveAttributeToColumn relationship does not leave any traces result for any of the variables. Indeed, as shown in Table 1, although during the preliminary phase tuples were obtained from both domains, these ones do not corresponds to the same variable at domains checkonly and enforce simultaneously.

Later analysis phase.

The final phase of the algorithm, called post-analysis, receives as input the tuples generated during the first stage (previous analysis) which could not be derived as direct traces due to the lack of trace information to establish the corresponding relationships between variables. Such missing information must be found in the post-conditions section of the transformation rule, at the *Where* clause.

The post-analysis stage is about studying the statements within the *Where* clause, to determine the traceability relationships between model elements specified by the variables. Following our example of the PrimitiveAttributeToColumn relation, we see that *Where* clause sentences allow us to finish associate those variables that after the direct trace determination phase seemed to be isolated. The assignment statement of variable *cn*, dependent on the outcome of if-then-else clause (Figure 17), allows us to establish its relationship with the variable *an* (5) (6), so we can infer the following conditional trace:

uml-Name::UmlAttribute⊕prefix+'_'+umlName::UmlAttribute→rdbmsType::RdbmsColumn

Next, the assignment statement of variable *sqltype* allows us to associate with the variable *pn*, determining the trace as follow:

PrimitiveTypeToSqlType(umlName::UmlPrimitiveType)→rdbmsType::RdbmsColumn

Constant traces.

The constant trace inference carried out by the presented mechanism is an exceptional case in the variable-based analysis. Although the detection of such traces is performed by the same component that implements the traceability strategy, it does not correspond to an analysis of variables. Indeed, since it is possible to infer a constant trace, direct or indirect, in an expression without variables, detection does not require the study of variables. The detection of this kind of traces is performed the same way to the others. In the case of direct constant traces, which are those that can be inferred after the pre-analysis, detection happens when it is verified that the expression of the tuple generated in an enforce domain has assigned a constant value. We have restricted the possibilities to two types of constants:

- Numeric, of integer values.
- String of characters.

Thus, the presence of an expression *model_element* = *k* in an enforce domain, where *k* is a numeric or character constant, will result in a trace *k*→ *model_element::Entity*, being *Entity* the class to which attribute *model_element* belongs. For indirect constant traces, which are those that are inferred in the post-analysis phase from sentences in the Where clause of the relationship, the situation is slightly different: instead of giving to a source model element a constant value in an enforce domain, this element is assigned a variable which is then defined in the post-conditions section of the rule, directly or through a conditional statement, as appropriate.

5 Comparison with other approaches

In order to assess the content of the proposal, we present a comparative study with related work that were introduced in Section 2.2. First we analyze the proposal of loosely coupled traceability for ATL by Jouault et al. [7], and then continue with the scheme of Grammel et al. [4] of traceability data extraction based on facets.

The purpose of the comparison is to show how each approach has addressed the problem of obtaining traceability information, and contrast differences and similarities of the works analyzed with the proposal itself. The comparison is based on two critical points: first, the proposed traceability metamodel, i.e. the way each proposal represents the traces, and second, in the mechanism of obtaining traceability information implemented by each approach.

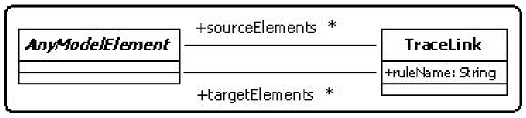


Fig. 18. Jouault's traceability metamodel

5.1 Loosely coupled traceability approach

The Jouault's proposal was one of the first works to generate automatic traceability information in the context of model-driven development. It is a reference work that shows a real model-driven solution to the problem of getting traceability information in model transformation. For these reasons, it has been chosen to compare against the study presented in this paper.

Traceability metamodel.

The work proposes a simple traceability metamodel, composed by a class called TraceLink, which contains an attribute that stores the name of the rule that generates the trace, and maintains two collections of AnyModelElement type objects, named *sourceElements* and *targetElements* (Figure 18). Such collections store source and target elements related by the application of the rule. The class AnyModelElement is abstract, and obviously depends on metamodel application environment.

Traceability information generation.

This approach suggests obtaining traces by adding extra code to ATL transformation rules, and also adding a trace output model that allows that traceability information be captured at the time of transformation execution. This modification does not alter the logic of the program, but adds additional content to the definition of the transformation. Moreover the authors present a scheme, also supported by the same transformation language, that allows extra code to be automatically added to the defined rules, avoiding doing that manually.

As an example, we present the definition of a simple transformation of a source model A, with a single character string attribute called name, in a destination model B , with a unique name attribute as well. Figure 19a shows the original code, which specify how transformation operates. This code is then modified to support traceability. Figure 19b shows the new version of the code. Although straightforward, this example allow us to make the following observations:

- Obtaining traceability information is implemented at the transformation level. Requires execution to generate the traces.
- Does not affect the logic of the definition, but adds additional information that blurs the legibility of the original transformation.
- It is absolutely independent of the models involved in the transformation.
- Uses a generic traceability metamodel, adaptable to other possible schemes.
- The automatic rules conditioning process for traceability (called TracerAdder) support can be done before compilation of the transformation.

```
module Src2Dst;
create OUT : Dst from IN : Src;
rule A2B {
  from
    s : Src!A
  to
    t : Dst!B (
      name <- s.name
    )
}
```

a. Original code

```
module Src2DstPlusTrace;
create OUT : Dst, trace : Trace from IN : Src;
rule A2BPlusTrace {
  from
    s : Src!A
  to
    t : Dst!B (
      name <- s.name
    ),
    traceLink : Trace!TraceLink (
      ruleName <- 'A2BPlusTrace',
      targetElements <- Sequence {t}
    )
  do {
    traceLink.refSetValue('sourceElements', Sequence {s});
  }
}
```

b. Modified code

Fig. 19. Jouault's traceability information generation

Facet-based traceability data extraction.

The Grammel work proposes a generic traceability framework that allows augmenting arbitrarily model transformation approaches with a traceability mechanism. It is a generic proposal designed to support traceability on an arbitrary number of model transformation approximations. The framework is based on a generic traceability interface (GTI Traceability Generic Interface) which provides the connection point for some arbitrary transformation languagesand provides an API to connect with traceability engine. The framework also defines a domain specific language called Trace-DSL which essentially determines what type of traceability information is interchangeable between the generic interface and traceability engines connectors.

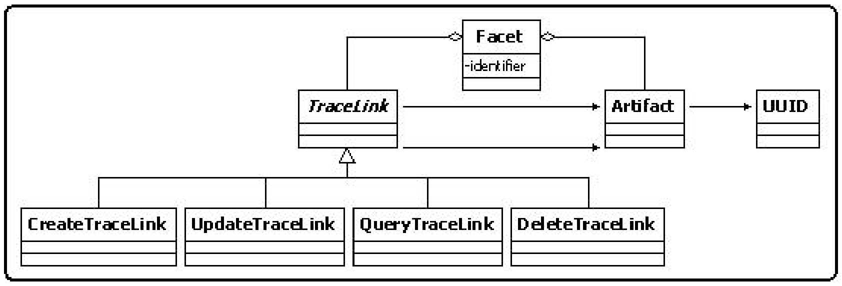


Fig. 20. Trace-DSL language fragment

Traceability metamodel.

The Grammel scheme is very interesting given that it adopts a completely different approach compare to that most authors. We will discuss a fragment of domain specific language Trace-DSL developed to support traceability (Figure 20). In this model, the traces are represented by the TraceLink component, which is considered an abstraction for the transition from one artifact to another. Every Artifact object is unambiguosly individualized by a Universal Unique Identifier (UUID), and represents any traceable product generated in the process of development, as a requirement or class, or a compound artefact, e.g. a method inside a class. A transition is always directed, therefore a from-to relation between artifacts is created by a trace link between source and destination artefacts. Whereas the traceability as tracking of all changes possibly applied to model elements during a transformation, Grammel et al. propose to breake down the model transformation chain into its elementary operations (eo_s) and to define for each type of *eo* a certain link type between the corresponding source and target model elements. According to these considerations, the author proposes the following types:

- CreateTraceLink, for a newly created target model elements.
- UpdateTraceLink, for an update (destructive or extension-only) as well as update in-place transformation.
- DeleteTraceLink, for a delete operation on a model element.
- QueryTraceLink, for a query (read) operation on a model, returning a subset of model elements.

Furthermore, to assign types to artifacts and traces, the proposal uses the concept of facet, where the Trace-DSL assigns a set of facets to every artifact and trace link, simplifying type hierarchy and seeking to provide extensibility to the proposed metamodel.

Traceability information generation.

As we have seen, the use of connectors allows interaction with eventually any model transformation language, arbitrarily. In particular, and considering that our proposal QVTrace has been defined in the context of QVT, we focus on determining how to work out this mechanism to get traceability information in transformations written for that language. The QVT connector developed by Grammel is written in a QVT subset called Operational Mappings. This language allows to define transformations using an imperative approach or complement transformations written in QVT Relations language with imperative operations (hybrid approach), when it is difficult to provide a fully declarative specification of a relationship, as well. Each relation defines a class that will be instantiated for the trace between model elements being transformed, and has a unique mapping with an operation that the operational mapping implements.

Under this scheme the traceability information extraction occurs in two steps: first, the model transformation is executed allowing the conversion of any instance of internal traceability metamodel QVT in a Trace-DSL instance by defining an operational mapping between metamodels. The second step is to import the Trace-DSL instance generated in the corresponding repository. This description allows us to observe the following:

- As with the Jouault's proposal, it operates at transformation level, i.e. requires the execution of the transformation to generate the traces.
- Allows arbitrary interaction with model transformation languages, although depends strongly on connector that enables connection to the transformation engine.
- In the context of QVT, use the Operational Mappings language.
- Defines an interesting traceability metamodel typifying different tracing scenarios (generation of new items, update, delete and query).
- It is independent of the models involved.
- Do not add any additional information that might alter patterns or logic of transformation.

5.2 Summary of main differences

After discussing the details of the two selected reference works and analizing the characteristics of the proposed approach, we will contrast the solutions according to the criteria defined above.

Traceability metamodel.

As described in previous sections, we have seen three different traceability metamodels proposals. First, a very generic Jouault's scheme, simple, flexible, whose emphasis is on maintaining traceability information from the viewpoint of the relationship between elements of source and target models. Second, we review Grammel's metamodel with a scenario approach that characterizes different types of trace links based on the impact on the target model. Finally, our QVTrace's scheme, closer to Jouault's approach, although more restrictive regarding the participants of the trace relationship and focused on the meaning of relationship, not just its members, i.e. the way of a source model element becomes in a the target model one.

While we can continue listing differences, each metamodel has its strengths and weaknesses, and they arise from the application and context in which they are defined. Clearly there is no “the” traceability metamodel, but different approaches that emphasize specific aspects depending on the use and results desired.

Traceability information generation.

If traceability metamodel defines what kind of traceability information obtained, the second comparison criterion considers the how to get it. As we have seen, the mechanism of Jouault is purely driven by models, defined with the same transformation language tools, automated, implemented at the level of transformation that adds extra information to the definition of this transformation, altering not logic but the readability of it. In contrast, Grammel's work is based on a DSL (Domain Specific Language) developed ad hoc, generic, adapted to an arbitrary number of transformation languages, although dependent on the capabilities and tools that it can provide for the construction of a fundamental connection component, implemented at the level of transformation. Finally, the proposal of these authors suggests that from the knowledge of the definition of a models transformation, and the models involved, it is possible to identify certain patterns or language constructs that allow to recognize traces, independently from the transformation process, characterizing not only the participants of the same, but the meaning or form, and does not alter in any way models nor transformation logic, automated, implemented outside the possibilities of model transformation language, but built with a major development environments used under the model-driven paradigm.

6 Conclusions and future work

Throughout the present paper we have reviewed the highlights of the model-driven development paradigm (MDD) and addressed the concept of traceability as a desirable feature in any model transformation. In this context, we have proposed an analysis technique based on variables that allows to identify certain patterns in a model transformation definition, written in QVT Relation language, that enable automatic trace inference. Besides being a theoretical proposal, this idea has been implemented in a prototype called QVTrace, which is designed as an Eclipse plugin to interact with other model-driven development tools designed for such framework.

In addition, we have detailed the main features of QVTrace and performed a comparative analysis with two schemes of similar kind: the proposed loosely coupled traceability of Jouault, and the generic traceability framework for facet-based traceability data extraction of Grammel et al. This study allow us to conclude that there is no ultimate traceability metamodel but it is strongly tied to the implementation of both trace generation mecanism and the information you want to keep, i.e, the application context. Unlike the work of other authors, our traceability inference proposal is completely independent of the transformation of models and works on the definition of it in the belief that it contains the whole traceability information implied.

Furthermore we can think about QVTrace as a traceability framework, that allows to decouple the process of obtaining traceability information from transformation execution. QVTrace proposes a traceability framework for model transformation integrated with the development environment most used by the community MDD, Eclipse. With an architecture madeup of a module reader/scanner of models, a module reader/analyzer of transformations, and a component that analyzes and processes this information, named TraceAnalizer, allows the developer to obtain model level traces, enabling its use as debbugging tool to define transformations, allowing to obtain information on the target model, prior to the completion of the transformation, and secondly, that it can be used to verify and ensure consistency between the source and destination models, in case the latter is modified outside the natural flow the paradigm suggests.

Finally, it is important to mention as possible future research to determine new constructions or trace cases to infer traceability information so far unidentified. A second point of interest would be finding the limits of the possible types of traces that the technique can generate, and to prove it formally.

References

1. mediniQVT. http://projects.ikv.de/qvt.
2. N. Aizenbud-Reshef, B. T. Nolan, J. Rubin, and Y. Shaham-Gafni. Model Traceability. *IBM System Journal*, 45(3):515-526, 2006.

3. Frank Budinsky, Stephen A. Brodsky, and Ed Merks. *Eclipse Modeling Framework*. Pearson Education, 2003.
4. Birgit Grammel and Stefan Kastenholz. A Generic Traceability Framework for Facet-based Traceability Data Extraction in Model-driven Software Development. In *Proceedings of the 6th ECMFA Traceability Workshop*, ECMFA-TW '10, pages 7-14, Paris, France, 2010.
5. Object Management Group. *MOF 2.0 Query/Views/Transformations RFP*, OMG document edition, October 2002.
6. F. Jouault and I. Kurtev. Transforming models with ATL. In *Satellite Events at the MoDELS 2005 Conference*, volume 3844 of Lecture Notes in Computer Science, pages 128-138, Berlin, 2005. Springer Verlag.
7. Frédéric Jouault. Loosely Coupled Traceability for ATL. In *Proceedings of the European Conference on Model Driven Architecture (ECMDA) workshop on traceability*, pages 29-37, Nuremberg, Germany, November 2005.
8. Anneke G. Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.
9. OMG. Meta object facility (MOF) 2.0 Query/View/transformation specification version 1.0. http://www.omg.org/spec/QVT/1.0/PDF/, April 2008.
10. The Institute of Electrical and Electronics Engineers. *IEEE Standard Glossary of Software Engineering Terminology*. New York, USA, September 1990.
11. Bert Vanhooff, Stefan Van Baelen, Wouter Joosen, and Yolande Berbers. Traceability as input for model transformation. In *Proceedings of the European Conference on Model Driven Architecture (ECMDA) workshop on traceability*, Haifa, Israel, June 2007.