Hundhausen, C. et al. (Eds.) (2010). *2010 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). Leganés-Madrid, Spain 21-25 September 2010: Proceedings.* IEEE, 119-126.
DOI: http://dx.doi.org/10.1109/VLHCC.2010.25

A Visual Specification Language for Model-to-Model Transformations

Esther Guerra
Computer Science Department
Universidad Carlos III de Madrid
Madrid, Spain
eguerra@inf.uc3m.es

Juan de Lara
School of Computer Science
Universidad Autónoma de Madrid
Madrid, Spain
Juan.deLara@uam.es

Dimitris Kolovos, Richard Paige
Computer Science Department
University of York
York, UK
{*dkolovos, paige*}*@cs.york.ac.uk*

***Abstract*—Model Driven Engineering promotes models as the core assets of projects and hence model transformations become first-class citizens in this approach. Likewise, the development of large scale transformations necessitates a systematic engineering process and supporting *modelling* notations. However, although many languages have been proposed to *implement* transformations, few allow their *specification* at a higher level of abstraction.**

In this paper we present a visual, formal, declarative specification language to express model-to-model transformations and their correctness properties. The language supports the two main approaches to model-to-model transformation – trace-based and traceless – with a unified formal semantics. Moreover, we provide a compilation of specifications into OCL as this has many practical applications, e.g. it allows injecting assertions and correctness properties for automated testing of transformation implementations based on OMG standards.

***Keywords*-model-driven engineering; model-to-model transformation; specification languages; transformation testing**

I. Introduction

Model Driven Engineering (MDE) is a software engineering approach that seeks increasing productivity and quality by raising the level of abstraction at which engineers work. For this purpose, models (in contrast to programs) are key assets in the development, and hence model transformations are the pillars of the process. A model transformation receives one input model and produces one output model, in the simplest case. If both models conform to the same meta-model the transformation is called endogenous, whereas if the meta-models are different it is called exogenous or model-to-model (M2M) transformation [1].

In order to become useful in industrial practice, engineers need methods and tools to analyse, design, implement and test complex and large M2M transformations. However, although many languages have been proposed to *implement* transformations [1], [2], [3], there is a lack of methods, notations and tools to cover further stages of the complete transformation life-cycle.

In standard software development, specification languages are commonly used to express desired properties about the applications to be built [4]. They focus on what the application should do without stating how to do it. Hence, they are closer to the system analysis (which could also be refined into design) than to its actual implementation. Formal specification languages like Z [4] or Alloy [5] have a mathematical underpinning that allows formal reasoning, refinement, proof, and specification-based testing of implementations [6].

In this paper we propose a high-level, formal, visual, declarative language to *specify* M2M transformations. Its purpose is not to implement transformations, but to express *what* the transformation is to do (but not *how*), as well as properties that transformed models should satisfy. In this sense, the role of our language for transformations is similar to the role of Z for software: providing support to the analysis and design of transformations. The language provides constructive and non-constructive primitives to specify relations that should hold between the input and output models, or forbidden situations. It also supports the two usual approaches to M2M transformation: *trace-based*, where explicit mappings define relations between the input and output models (as e.g. in QVT-Core [7] and triple graph grammars [8]), and *traceless* (as e.g. in QVT-R [7], ATL [2] and ETL [3]). Specifications can be used in two ways: (i) as a functional, potentially loose, definition of (part of) the expected behaviour of transformation implementations; and (ii) to provide correctness properties of the transformation.

Fig. 1 outlines our approach. First, the transformation designer uses our specification language to define the transformation behaviour, verification properties, and requirements on the valid input models (label 1). This specification has a formal semantics and can be analysed to discover redundancies, contradictions, and to measure coverage of the involved languages (label 2). Next, the developer uses the specification as a high-level model to implement the transformation (label 3). This implementation is tested by injecting assertions automatically derived from the specification (label 4). Assertions act as an *oracle* describing structural invariants that output models should satisfy, and are used for automated testing (labels 5, 6). They are also used to test whether a model can be used as input for the transformation.

Altogether, the contributions of this paper are the following. We propose a novel, visual specification language for M2M transformation, supporting both trace-based and

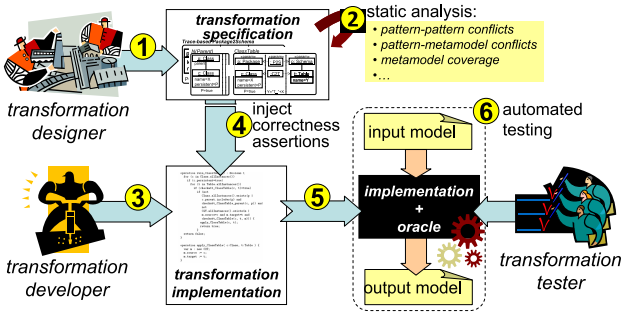


Figure 1. Scheme of our approach.

traceless styles. This language can be used in initial stages of the transformation development cycle (analysis and design) and enables the automated verification of implementations. To the best of our knowledge, no such language has been proposed before. We also provide a compilation of specifications into OCL. This enables the injection of correctness assertions in order to automate testing of implementations for QVT and other languages based on OMG standards, such as ATL or ETL. Since complex, large-scale transformations are frequently encoded using textual languages, we aim at keeping the best of visual and textual transformation languages. Finally, we report on an Eclipse-based prototype, and illustrate the injection of OCL assertions for testing ETL transformations.

Paper organization. Section II presents the syntax and formal semantics of our language. Section III shows its compilation into OCL. Section IV describes tool support and an example. Finally, Section V discusses related research and Section VI concludes.

II. A M2M Specification Language

Our language is used to test implementations, but it is independent of them. It supports both trace-based and traceless styles of specification, which allows one to express properties for implementation languages that use an explicit handling of traces (e.g. QVT-Core, TGGs), and also for languages that do not make use of them (e.g. QVT-R, ATL, ETL). The first style is closer to implementation, since it implies creating traces for each transformed element and its targets. Traceless specifications do not use traces, but a mechanism to express pre- and post-conditions which may refer to other parts of the specification. For instance, QVT-R uses *when* and *where* constructs to define pre- and post-conditions, respectively. Interestingly, both styles share similar semantics and can be formalized in a unified framework.

A. Constraint triples

A specification in our language is made of patterns. Here we extend our theory developed in [9] for the definition of both trace-based and traceless specifications. Patterns are based on the concept of triple graph [8] to represent the input, output and trace models (called source, target and correspondence). A triple graph $G = \langle G_S, G_C, G_T, cs, ct \rangle$ is made of two graphs G_S and G_T called *source* and *target*, related through a *correspondence* graph G_C and two graph morphisms $cs\colon G_C \to G_S$ and $ct\colon G_C \to G_T$. For trace-based patterns, G_C contains the traces between nodes in G_S and G_T, while for traceless patterns, G_C is empty.

We use *symbolic graphs* [10] to describe the structure of the three graphs. Symbolic graphs are typed and have attributed nodes and edges, but instead of having a possibly infinite set of data values, they use a finite set of sorted variables ν, and a formula α constraining the allowed values for these variables. Thus, *constraint triples* have the form $C = \langle G, \nu, \alpha \rangle$, where G is a triple graph whose data nodes are replaced by variables in ν. We use constraint triples to represent both usual models (called *ground constraints*, where α restricts the attributes to take exactly one value), as well as constraints to be satisfied by models.

Example. Fig. 2 shows examples of trace-based and traceless constraints, modelling part of the class-to-relational transformation [7]. They relate persistent UML packages with RDBMS schemas. The traceless constraint does not show the correspondence graph as it is empty. In both cases the formula α is shown at the bottom, we omit the conjunctions between terms, and place in the left compartment the terms containing only variables from the source graph, in the right compartment the terms containing only variables from the target, and in the middle the terms containing variables of both graphs. Note that "=" denotes equality, not assignment. We can use any logic for α, but here we use first-order logic with OCL-like syntax.

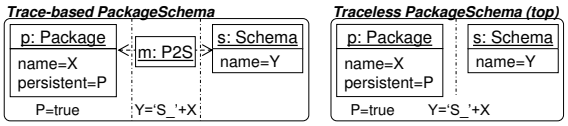


Figure 2. Trace-based and traceless constraints.

Constraint triples are related through C-morphisms $a\colon C_1 \to C_2$, made of a triple graph morphism with the following conditions: the formula α_2 of C_2 must imply the formula α_1 of C_1, and the same implication is demanded for the source and target restrictions ($\alpha_2|_S \Rightarrow \alpha_1|_S$ and $\alpha_2|_T \Rightarrow \alpha_1|_T$). Roughly, the source restriction $\alpha|_S$ (resp. target $\alpha|_T$) of a formula is the formula α but considering the variables of the source (resp. target) graph only [9]. The source restriction $C|_S$ of a constraint triple C is made of the source graph and the formula $\alpha|_S$, and similar for the target restriction.

B. Trace-based Patterns

We use the previous concepts to build trace-based patterns. We define two kinds of pattern with same structure

but different interpretation: positive and negative (called P-patterns and N-patterns). A pattern is made of a main constraint triple Q (to be satisfied in the case of P-patterns, and forbidden to occur in the case of N-patterns), and may contain a positive pre-condition C and a set of negative pre-conditions N_i.

Def. 1 (Trace-based pattern)**.** *A trace-based pattern* $P = \langle C \xrightarrow{q} Q,\ N_{pre} = \{n_i\colon Q \rightarrow N_i\}_{i\in I}\rangle$ *consists of a main constraint triple* Q*, a (possibly empty) positive pre-condition* C*, a* C*-morphism* q*, and a set* N_{pre} *of negative pre-conditions.*

Example. Fig. 3 shows our concrete visual syntax for a trace-based P-pattern, which contains a main constraint (named *ClassTable*), a negative pre-condition (denoted by *N(Parent)*), and a positive pre-condition (annotated on the main constraint with $\langle\langle param\rangle\rangle$). The pattern states that each persistent class should be related to a table, when the class has no parent (negative pre-condition) and if the class' package is mapped to a schema (positive pre-condition).

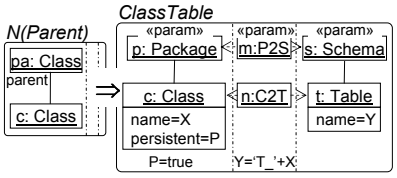


Figure 3. Example of trace-based P-pattern.

Fig. 4 shows an N-pattern. While the P-pattern can be used to specify constructively a transformation, this N-pattern expresses an invariant, i.e. a verification property reflecting the beliefs of the designer about the properties that should hold in all related models. The N-pattern states that if a class has no two attributes with same name (negative pre-condition *N(AttrDup)*) then the associated table should not have duplicated columns (main constraint *N(ColDup)*). This property is indeed false, if attributes of children classes are stored in the same table, and classes can redefine attributes.

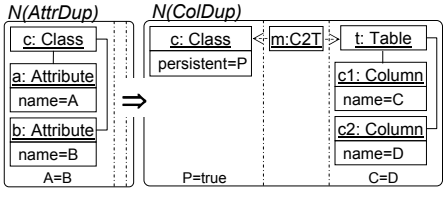


Figure 4. Example of trace-based N-pattern.

Sometimes, M2M transformations are not designed to cope with every valid source model, but to work with a subset of models of the source language. Our patterns can also be used to explicitly express the conditions that we ask the source models to qualify for the transformation. As an example, Fig. 5 shows an N-pattern that forbids attribute redefinition (operation *ancestors* returns the set of ancestors of a given class). Similarly, we can also use patterns to specify properties that any output model of the transformation should fulfill.

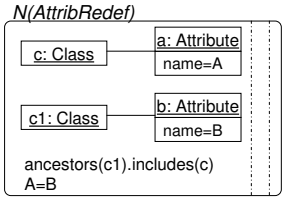


Figure 5. N-pattern constraining the source language.

In M2M transformation, we are interested in knowing whether a target model is a correct translation of a source model, or vice-versa. For this purpose we interpret patterns either source-to-target or target-to-source (forwards/backwards). In the former case, we check that each forward-enabled pattern is actually satisfied, and similar for the backward case. If two models satisfy the patterns both forwards and backwards, we say that they are *synchronized*. We start by defining enabledness of a pattern for the forward case; the backward case is symmetrical.

Def. 2 (Forward pre-condition)**.** *Given a pattern* P*, its forward positive pre-condition* $F^+(P) = C +_{C|_S} Q|_S$ *is given by the pushout of its positive pre-condition* C *and the source restriction of the main constraint* Q*, while its set of forward negative pre-conditions is* $F^-(P) = \{q^S\colon F^+(P) \rightarrow N_i^S,\ with\ N_i^S = C +_{C|_S} N_i|_S\}_{i\in I}$.

Example. Fig. 6 shows the forward positive pre-condition of pattern *ClassTable*, $F^+(ClassTable)$, which results from merging C (objects `p`, `s` and `m`) and $Q|_S$ (objects `p`, `c` and their link) through $C|_S$ (object `p`). This is called a pushout in category theory. In our case, pushouts are made like in triple graphs, and then taking the conjunction of the formulae [9]. The pattern has one forward negative pre-condition N_1^S, depicted to the left.

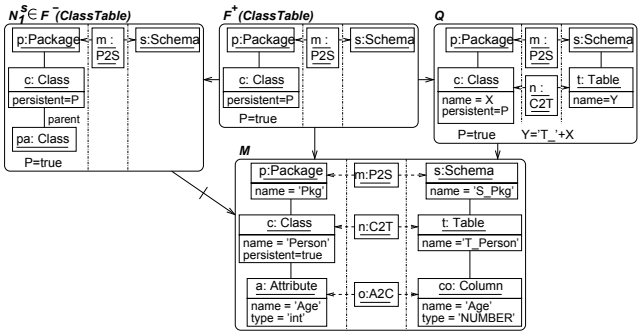
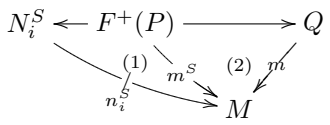


Figure 6. Trace-based forward satisfaction example.

A pattern P is forward-enabled in a constraint triple M (not necessarily ground) if an occurrence of its forward positive pre-condition $F^+(P)$ is found in M, and no occurrence of its negative forward pre-conditions is found. A P-pattern (N-pattern) is satisfied at an enabled match, if the match can be (cannot be) extended to the pattern's main constraint Q.

Def. 3 (Forward enabledness)**.** *Given pattern P and constraint triple M, P is forward-enabled at match $m^S\colon F^+(P) \to M$, written $M \vdash_{m^S,F} P$, iff $\forall i \in I, \nexists n_i^S\colon N_i^S \to M$ s.t. (1) commutes in the diagram below.*



Def. 4 (Forward satisfaction)**.** *Given pattern P, constraint triple M and $M \vdash_{m^S,F} P$, P is forward-satisfied at m^S, written $M \models_{m^S,F} P$, iff $\exists m\colon Q \to M$ s.t. (2) commutes in the diagram above if P is a P-pattern, or iff $\nexists m\colon Q \to M$ if P is an N-pattern. P is forward-satisfied in M, written $M \models_F P$, iff $\forall m^S$ s.t. $M \vdash_{m^S,F} P$, $M \models_{m^S,F} P$.*

Example. The pattern in Fig. 6 is forward-enabled in one occurrence, the one identifying objects `p`, `m`, `s` and `c` in $F^+(ClassTable)$ and M, as the forward negative pre-condition N_1^S is not found in M. The pattern is actually forward-satisfied by M because this occurrence can be extended to Q.

Specifications are conjunctions of patterns, hence M forward-satisfies a specification S ($M \models_F S$) if it forward-satisfies all its patterns. Two models are synchronized if each other is a correct forward/backward translation of the other: $M \models_F S$ and $M \models_B S$.

C. Traceless Patterns

Similar to QVT-R, patterns in the second style of specification do not make use of traces, but provide constructs to check if other patterns in the specification are satisfied (*when* clause, a pre-condition), or to demand the satisfaction of other patterns (*where* clause, a post-condition). Therefore they need a way to express dependencies between patterns. As for trace-based specifications, we consider P- and N-patterns having the same structure, although for N-patterns we demand $where = \emptyset$ (we cannot ask for additional conditions on a non-existing occurrence of Q). As a difference from trace-based patterns, we distinguish between *top* and *non-top* patterns. The former must be satisfied always, and the latter only when invoked from the *where* clause of other patterns. Recall that we use the same underlying structure as for trace-based patterns, but in this case the correspondence graph is not shown because it is empty.

Def. 5 (Traceless pattern)**.** *A traceless pattern $R = \langle Q, N_{pre} = \{n_i\colon Q \to N_i\}_{i\in I}, when, where, top\rangle$ is made of a main constraint triple Q, a set N_{pre} of negative pre-conditions, two sets* when *and* where *of dependencies for Q, and a boolean flag* top.

Example. Fig. 7 depicts three traceless patterns for the specification of the class-to-relational transformation. Pattern *ClassTable* is top and demands a table for each class without parents (negative pre-condition *N(Parent)*). The *when* clause makes this necessary only if the class' package and the table's schema satisfy the *PackageSchema* pattern (shown in Fig. 2). Moreover, if this is the case, then both patterns *AttributeColumn* and *ParentClassTable* should be satisfied for the class and table. While the former demands pairs of attributes and columns in the given class and table, the latter descends recursively through the inheritance hierarchy demanding the satisfaction of *AttributeColumn* at each child class.

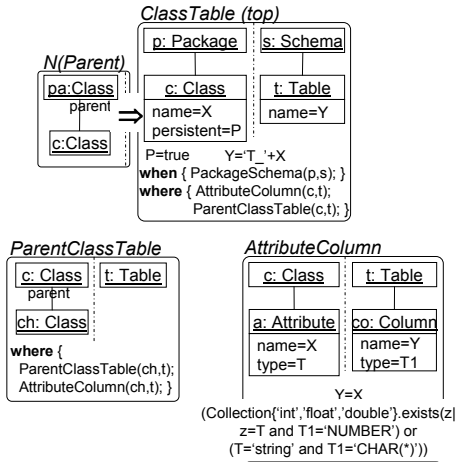


Figure 7. Traceless patterns.

The use of *when* and *where* clauses creates dependencies between patterns. In particular, given the main constraints Q_1 and Q_2 of two patterns, a dependency is given by $Q_1 \xleftarrow{d_1} D \xrightarrow{d_2} Q_2$, where D contains the elements passed as parameter in a *when* or *where* clause relating them. Thus, similar to the forward pre-condition notion for trace-based patterns, we define forward dependencies for traceless patterns generalizing the pushout construction in Def. 2 to an arbitrary number of dependencies (not just one). Then we take the amalgamation of all of them.

Def. 6 (Forward dependency)**.** *Given a traceless pattern R and a dependency $Q \xleftarrow{d} D$, the forward positive dependency is given by $F_d^+(R) = F^+(\langle D \to Q, N_{pre}\rangle)$, while the set of forward negative dependencies is given by $F_d^-(R) = F^-(\langle D \to Q, N_{pre}\rangle)$, see Def. 2.*

Given R and a dependency set $DS = \{Q \xleftarrow{d_j} D_j\}_{j\in J}$, the forward positive dependency is given by $F_{DS}^+(R) = W$ as shown to the left of Fig. 8, with I the limit of $\{p_j\}$,

W the colimit of $\{i_j\}$, and u exists due to the limit universal property. The set of forward negative dependencies is $F^-_{DS}(R) = \bigcup_{d_j \in DS} \{W \to N_i^W\}_{i \in I}$, with N_i^W a pushout calculated as shown to the right of Fig. 8.

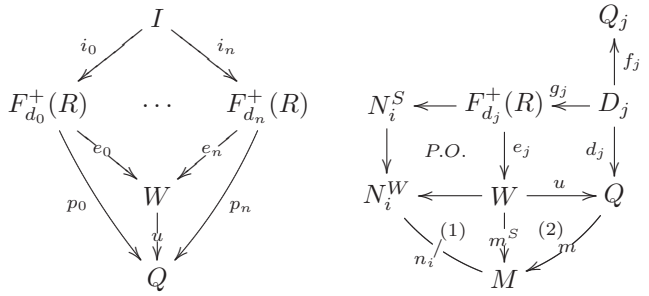


Figure 8. Forward dependency set.

Next we define the conditions for a traceless pattern R to be forward-enabled. As in the trace-based case, we have to find an occurrence of $F^+(R)$ and no occurrence of $F^-(R)$, but here there is no positive pre-condition but a set of *when* dependencies. Thus, for a traceless pattern to be forward-enabled, we build $F^+_{when}(R)$ and demand each *when* dependency to be satisfied.

Def. 7 (Forward enabledness)**.** *R is forward-enabled in a constraint triple M at match $m^S\colon W \to M$ (with $W = F^+_{when}(R)$), written $M \vdash_{m^S,F} R$, iff $\nexists n_i\colon N_i^W \to M$ with (1) commuting to the right of Fig. 8, and $\forall Q_j \xleftarrow{f_j} D_j \xrightarrow{d_j} Q \in when$, $SAT^F(R_j, m^S \circ e_j \circ g_j, f_j)$ (see Def. 8).*

Example. Fig. 9 shows a constraint M where the pattern *ClassTable* is enabled: there is an occurrence of $W = F^+_{when}(ClassTable)$ (made of objects `p`, `s` and `c`) that satisfies the pattern $PackageSchema$ for the commuting dependency $F^+_{f_1}(PackageSchema)$. The forward positive dependency $F^+_{f_1}(PackageSchema)$ is calculated taking $D_1 \xrightarrow{f_1} Q_1$. For simplicity we have omitted the negative pre-condition, which avoids the pattern to be enabled in class `c2`. We demand non-cyclic dependencies between patterns as otherwise we may obtain an infinite loop when testing the *when* clause.

We define the forward satisfaction of traceless patterns using a predicate SAT^F with three parameters: (1) the pattern R to be checked, (2) a morphism $D \to M$ with which its forward positive dependency $F^+_{when}(R)$ has to commute, and (3) a dependency $D \to Q$, which may come from a caller *where* section, and is actually treated as an additional pre-condition in the *when* clause. In this way, the predicate may demand the satisfaction of other patterns at certain matches that are passed as parameters from invoking *when* or *where* clauses, the former coming from Def. 7, and the latter from recursive calls in Def. 8.

Def. 8 (Forward satisfaction)**.** *Given a P-pattern R, predicate $SAT^F(R, m_D\colon D \to M, d\colon D \to Q)$ holds iff: $\forall m^S \in \{m^S\colon W \to M \mid m_D = m^S \circ e,$ with $W = F^+_{when \cup \{d\}}(R), M \vdash_{m^S,F} R, D \xrightarrow{e} W\}$, $\exists m\colon Q \to M$ s.t. (2) commutes to the right of Fig. 8, and $\forall Q_k \xleftarrow{f_k} D_k \xrightarrow{d_k} Q \in where$, $SAT^F(R_k, m \circ d_k, f_k)$.*

If R is an N-pattern, everything is the same, but we demand the non-existence of $m\colon Q \to M$ s.t. (2) commutes to the right of Fig. 8 (and nothing else as $where = \emptyset$).

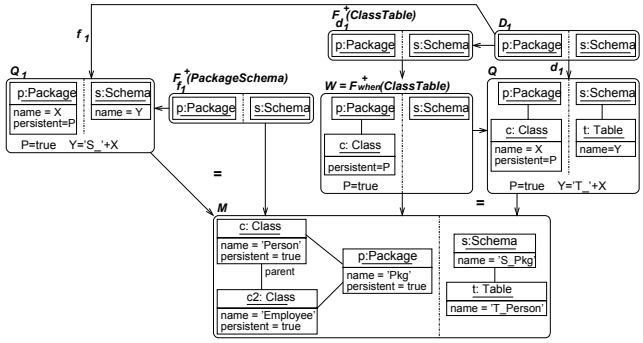


Figure 9. Traceless forward satisfaction example.

Example. The forward-enabled occurrence of pattern *ClassTable* in Fig. 9 is satisfied because we find an occurrence of the pattern's main constraint Q and the *where* dependencies are satisfied: (i) *AttributeColumn* is trivially satisfied as `c` has no attributes, and (ii) *ParentClassTable* is satisfied as we find one occurrence of it but the child class has no attributes.

The satisfaction of a traceless specification demands the satisfaction of all their top-level patterns.

Def. 9 (Specification forward satisfaction)**.** *Given a traceless specification S and a constraint triple M, $M \models_F S$ iff $SAT^F(R, \emptyset \to M, \emptyset \to Q)$ $\forall R \in S | R$ is top.*

Satisfaction of traceless specifications can be tested on traced models (i.e. triple graphs where the correspondence graph is not empty). This makes such specifications more independent of the implementation mechanism, which can be based on traces or not. On the contrary, trace-based specifications necessitate from traced models.

III. Compilation into OCL

In this section we provide a practical way for testing satisfaction of our patterns by their compilation into OCL (using the EOL syntax [11]). Our aim is generating invariants to automatically check the satisfaction of specifications by models, and which can be injected in the transformation implementations for testing purposes. We choose OCL because it is an OMG standard and can be integrated in transformation languages of widespread use, such as QVT, ATL or ETL. We start by showing the compilation of

traceless patterns, as the compilation of trace-based ones can be expressed in terms of the former.

For traceless patterns, we generate one set of operations from each (P- and N-) pattern, which only differ in their parameters. In particular, one operation is generated from each pattern call in a *when* or *where* clause, and one additional operation without parameters is generated for top patterns. We only show the compilation schema for the operation without parameters, since the others are built similarly (but omitting finding a match for the objects received as parameter). We assume just one pattern in the *when* and *where* clauses for readability reasons, and use the following notation:

- *p:* name of compiled pattern
- *when-p:* name of pattern in the when clause
- *where-p:* name of pattern in the where clause
- *when-p.param, where-p.param*: objects in the call to *when-p* and *where-p*, respectively
- *check-p(...)*: OCL expression that checks the graphical and attribute conditions imposed by *p* on the objects received as parameter
- *check-n(...)*: like *check-p*, but checks a negative pre-condition *n* instead of *p*

The scheme of the OCL code for checking the forward satisfaction of a traceless P-pattern is:

```
operation sat_p () : Boolean {
  return
  -- a) for each occurrence of objects a1,...,am
  --    in when-p.param that satisfy when-p
  a1.type.allInstances().forAll(a1 | ...
    am.type.allInstances().forAll(am |
      when-p(a1,...,am) implies
  -- b) for each occurrence of objects b1,...,bn
  --    in the source of p and not in when-p.param
      b1.type.allInstances().forAll(b1 | ...
         bn.type.allInstances().forAll(bn |
           check-p(a1,...,am,b1,...,bn)
  -- c) if it does not violate any negative pre-condition
  --    of p (being c1,...,co the objects in the negative
  --    pre-condition different from the a and b objects)
           and not
           c1.type.allInstances().exists(c1 | ...
             co.type.allInstances().exists(co |
               check-n(a1,...,am,b1,...,bn,c1,...,co)
  -- d) then there must be an occurrence of p (being
  --    d1,...,dp the objetcs in the target of p which
  --    are not in when-p.param)
           implies
           d1.type.allInstances().exists(d1 | ...
             dp.type.allInstances().exists(dp |
               check-p(a1,...,am,b1,...,bn,d1,...,dp)
  -- e) and satisfies where-p for the objects e1,...,eq
  --      in where-p.param (already matched by a, b and d)
               and where-p(e1,...,eq);
}
```

In the previous operation, fragments `a)`, `e)` and `c)` are omitted if the pattern has an empty when clause, an empty where clause, or an empty set of negative pre-conditions, respectively. The compilation schema for N-patterns is similar to that for P-patterns, but the existential operator in fragment `d)` is preceded by `not`. Finally, the compilation for backward satisfaction implies just substituting source by target (and vice-versa).

Example. The compiled code for the traceless pattern *ClassTable* is:

```
operation sat_ClassTable () : Boolean {
  return
  Package.allInstances().forAll(p | --------------- a)
    Schema.allInstances().forAll(s |
      PackageSchema(p,s) implies
      Class.allInstances().forAll(c | ------------- b)
        (p.class.includes(c) and c.persistent=true)
        and not ---------------------------------- c)
        Class.allInstances().exists(pa |
          c.parent.includes(pa))
        implies ---------------------------------- d)
        Table.allInstances().exists(t |
          s.table.includes(t) and t.name='T_'+c.name
          and AttributeColumn(c,t) ---------------- e)
          and ParentClassTable(c,t))))));
}
```

The compilation schema of trace-based patterns is much simpler: (i) only one operation without parameters is generated from each pattern; (ii) fragments `a)` and `b)` are merged so that the resulting fragment looks for all matches of the pattern pre-condition (i.e. all elements in the positive pre-condition and the source of the main constraint which satisfy the graphical and attribute constraints); and (iii) no fragment `e)` is generated. Note that in this case, the generated OCL conditions actually check that traces exist when they appear in a pattern, while for traceless patterns this is not so, so they are independent of the implementation mechanism.

Example The operation derived from the trace-based pattern *ClassTable* is:

```
operation sat_ClassTable () : Boolean {
  return
  Package.allInstances().forAll(p | ---------------- a+b)
    Schema.allInstances().forAll(s |
      Class.allInstances().forAll(c |
        P2S.allInstances().forAll(m |
          (p.class.includes(c) and c.persistent=true and
          m.source=p and m.target=s
          and not  ---------------------------------- c)
          Class.allInstances().exists(pa |
            c.parent.includes(pa)))
          implies ---------------------------------- d)
          Table.allInstances().exists(t |
            C2T.allInstances().exists(n |
              s.table.includes(t) and t.name='T_'+c.name
              and n.source=c and n.target=t))))));
}
```

As stated previously, this OCL code can be used in many ways. The next section shows an application to automated testing of transformation implementations.

IV. Tool Support and Example

We have built an Eclipse tool to define pattern specifications using a visual concrete syntax. It has been developed with GMF, and includes a code generator to synthesise EOL code [11] (an extension of OCL) for the chosen scenario (forwards/backwards, either for traceless or trace-based specifications). This code can be injected in ETL transformation implementations in two ways: (i) assertions coming from patterns expressing conditions on the source model, like the pattern in Fig. 5, are tested before executing

the transformation; (ii) patterns expressing expected properties of the target model, as well as verification or functional properties of the transformation, are tested after executing the transformation. Hence, given an input model, it is first checked if it qualifies for the transformation. If it does, the transformation is executed and the user is informed of the patterns that are or are not satisfied, and of the rules that should be revised.

Fig. 10 shows a verification traceless P-pattern defined in the tool. The pattern specifies how to handle multiple inheritance. In particular, it seeks two top-level persistent classes `c1` and `c2`, ancestors of a third class `c`. The fact that `c1` and `c2` are top-level is checked by the negative preconditions *NoAncestor1* and *NoAncestor2*, whereas the fact that `c1` and `c2` are ancestors of `c` is checked by operation `ancestors` in the formula. Then, for each attribute `a` of `c`, the pattern demands a matching column in both tables `t1` and `t2`. This is checked in the `where` section by calling the functional requirement pattern *AttributeColumn* for each table. Moreover, the `when` section checks that `t1` and `t2` are associated with `c1` and `c2` by calling *ClassTable2* (equal to *ClassTable* but without `where` section). An additional verification pattern checks that if a top-level class does not have attributes with same name (e.g. no redefined attributes in children), its associated table does not have columns with same name.

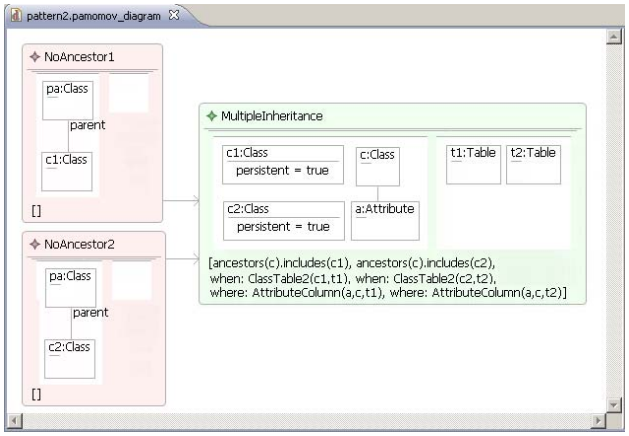


Figure 10. Verification pattern.

After defining the patterns with the functional requirements and verification properties, we can generate EOL code to verify a particular transformation implementation. Fig. 11 shows part of the ETL code that implements the forward transformation. The implementation is a refinement of the functional specification, as in addition it creates primary and foreign keys and considers object references. This implementation is incorrect because it does not consider multiple inheritance: when an attribute is translated into a column, the column is placed in the table associated to the top-most class (line `c.table ::= a.owner.getTopClass();`). However, the operation `getTopClass` assumes single inheritance and returns a unique class (and `::=` returns its associated table). Therefore, this implementation fails when tested with models having multiple inheritance, and is detected by our patterns, as the pop-up window in Fig. 11 shows. The feedback mentions the rules to be revised because these are annotated with the patterns they address (line `@patterns=...`).

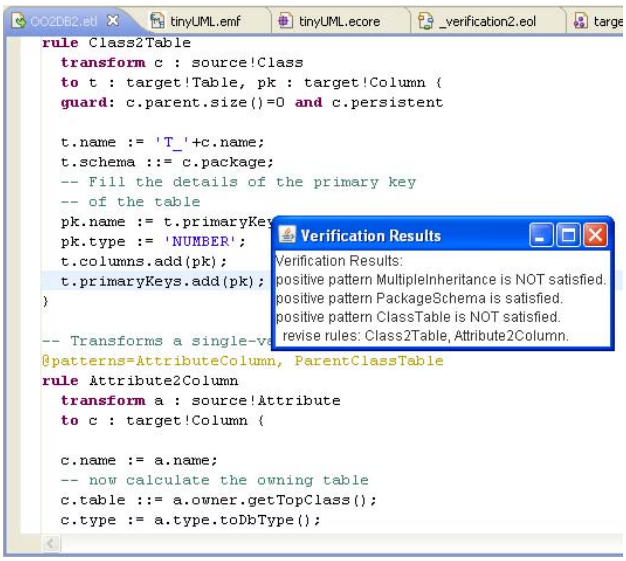


Figure 11. Testing an implementation.

It is interesting to note that a specification expresses requirements, and is independent of how the implementation actually performs its job. In our example, the implementation does not use recursion on children classes (like pattern *ParentClassTable* does), but a method to obtain the table of the top-most class. Second, we found it useful to classify patterns as functional or verification patterns, where the latter usually depend on the former. Third, functional patterns do not need to specify the behaviour of the complete transformation and cover all requirements, but only the most critical ones (in our example, it did not address primary or foreign keys nor references). Moreover, we do not even have to use the *same* meta-model for specification and implementation, but the meta-model of the implementation can be a refinement of the specification one. Finally, specifications are independent of the implementation language, and they can be used for testing implementations written in different languages. In particular, the approach is useful to test large textual implementations, and we used it for the run-time verification of a transformation of more than 1600 lines of code in the context of a European project.

V. Related Work

Our traceless language is inspired by QVT-R [7], but enriched with N-patterns (i.e. non-constructive primitives),

graphical negative pre-conditions and bidirectional attribute computations. Whereas QVT-R implementations are able to execute parts of the standard [7], we are working in execution support for functional patterns, but there are some issues. First, our attribute computations are bidirectional, which means doing either algebraic manipulation of formulae or using constraint solving when the transformation is given a direction. Bidirectional conditions like $X+Y=Z+V$, which involve variables of source and target elements, *are not* supported by existing QVT-R implementations (assignments are supported, but not general formulae). The non-constructive nature of N-patterns would also need constraint solving. Finally, specifications may be loose: a source model may have several correct target models. Implementations can *refine* this behaviour choosing deterministically one solution.

The formal semantics of our traceless language is immediately applicable to QVT-R. There are few attempts to give formal semantics to QVT-R. In [12], the authors compile simplified QVT-R into TGGs. In [13], a game-theoretic semantics for check-only QVT-R is given, but the semantics is given in an abstract way, neglecting issues like bindings, pattern matching and parameter passing. There are a few QVT-R concepts we do not cover yet though, like having arbitrary formulae in *when* and *where* instead of sets.

Even though there are many languages to implement transformations, very few works propose higher-level notations for transformation design [14]. To our knowledge, no language has been proposed for specification of implementation properties, as we do in this paper. Even though there are languages for expressing bi-directional transformations, they are unsuitable for their use as formal specification languages. Some of them, like QVT-R, have no formal semantics. Others, like TGGs, are based on rules and hence they are not suitable for testing, where a language based on constraints is more appropriate.

Finally, our work also contributes to the area of transformation testing by providing a language that simplifies the specification of oracles to automate the comparison of the actual and expected results of transformations, where current approaches require the manual specification of complex OCL constraints [15].

VI. Conclusions and Future Work

In this paper we have presented a high-level M2M *specification* language, its formal semantics, its compilation into OCL, tool support, and its application for M2M transformation testing. Concerning the latter, we have shown the benefits of a visual specification language to guard the correctness of large, textual transformation implementations. Moreover, our traceless language has a formal algebraic semantics applicable to QVT-R.

We are currently working on executability of specifications by combining transformation languages with constraint solvers. However, in some scenarios, implementations coded by hand may be more efficient or scalable. We are also working in the analysis of specifications, studying the strengths and equivalence of both styles of specification, and on methods to derive test cases from specifications.

Acknowledgment

Work funded by the Spanish Ministry of Science and Innovation through project TIN2008-02081 and mobility grants JC2009-00015 and PR2009-0019; and by the R&D programme of the Madrid Community, project S2009/TIC-1650.

References

[1] K. Czarnecki and S. Helsen, "Feature-based survey of model transformation approaches," *IBM Systems Journal*, vol. 45, no. 3, pp. 621–646, 2006.

[2] ATL, `http://www.eclipse.org/atl/`.

[3] D. S. Kolovos, R. F. Paige, and F. Polack, "The Epsilon Transformation Language," in *ICMT'08*, ser. LNCS, vol. 5063. Springer, 2008, pp. 46–60.

[4] J. M. Spivey, "An introduction to Z and formal specifications," *Softw. Eng. J.*, vol. 4, no. 1, pp. 40–50, 1989.

[5] D. Jackson, *Software Abstractions. Logic, Language, and Analysis*. MIT Press, 2006.

[6] G. Bernot, M. C. Gaudel, and B. Marre, "Software testing based on formal specifications: a theory and a tool," *Softw. Eng. J.*, vol. 6, no. 6, pp. 387–405, 1991.

[7] QVT, `http://www.omg.org/spec/QVT/1.0/`.

[8] A. Schürr, "Specification of graph translators with triple graph grammars," in *WG'94*, ser. LNCS, vol. 903. Springer, 1994, pp. 151–163.

[9] E. Guerra, J. de Lara, and F. Orejas, "Pattern-based model-to-model transformation: Handling attribute conditions," in *ICMT'09*, ser. LNCS, vol. 5563. Springer, 2009, pp. 83–99.

[10] F. Orejas, "Attributed graph constraints," in *ICGT'08*, ser. LNCS, vol. 5214. Springer, 2008, pp. 274–288.

[11] D. S. Kolovos, R. F. Paige, and F. Polack, "The Epsilon Object Language (EOL)," in *ECMDA-FA'06*, ser. LNCS, vol. 4066. Springer, 2006, pp. 128–142.

[12] J. Greenyer and E. Kindler, "Comparing relational model transformation technologies: implementing query/view/transformation with triple graph grammars," *Softw. and Syst. Mod.*, vol. 9, no. 1, pp. 21–46, 2010.

[13] P. Stevens, "A simple game-theoretic approach to checkonly QVT relations," in *ICMT'09*, ser. LNCS, vol. 5563. Springer, 2009, pp. 165–180.

[14] M. Siikarla, M. Laitkorpi, P. Selonen, and T. Systä, "Transformations have to be developed ReST assured," in *ICMT'08*, ser. LNCS, vol. 5063. Springer, 2008, pp. 1–15.

[15] J.-M. Mottu, B. Baudry, and Y. Traon, "Model transformation testing: oracle issue," in *ICSTW'08*, 2008, pp. 105–112.