Inter-Modelling: from Theory to Practice

Esther Guerra[1], Juan de Lara[2], Dimitrios S. Kolovos[3], and Richard F. Paige[3]

[1] Universidad Carlos III de Madrid (Spain), `eguerra@inf.uc3m.es`
[2] Universidad Autónoma de Madrid (Spain), `Juan.deLara@uam.es`
[3] The University of York (UK), {`dkolovos, paige`}`@cs.york.ac.uk`

Abstract. We define inter-modelling as the activity of building models that describe how modelling languages should be related. This includes many common activities in Model Driven Engineering, like the specification of model-to-model transformations, the definition of model matching and model traceability constraints, the development of inter-model consistency maintainers and exogenous model management operators. Recently, we proposed a formal approach to specify the allowed and forbidden relations between two modelling languages by means of bidirectional declarative patterns. Such specifications were used to generate graph rewriting rules able to enforce the relations in (forward and backward) model-to-model transformation scenarios. In this paper we extend the usage of patterns for two further inter-modelling scenarios – model matching and model traceability – and report on an EMF-based tool implementing them. The tool allows a high-level analysis of specifications based on the theory developed so far, as well as manipulation of traces by compilation of patterns into the Epsilon Object Language.

1 Introduction

Model Driven Engineering (MDE) attacks the accidental complexity in the software development process by increasing the abstraction level at which engineers work. Models (rather than code) are the core assets, and are used to generate code, validation and verification. Models are seldom oblivious of each other, and hence many activities in MDE involve building relations between two or more models either manually or (semi-)automatically. The development of systematic, well-founded techniques and tools for the creation and maintenance of inter-model relations is therefore at the core of MDE, and is especially critical in large-scale projects involving vast amounts of inter-related models [11].

The specifications of inter-model relations can be used in many ways. For instance, a model-to-model (M2M) transformation specification expresses how models of a language should be related with models of another one, and it is actually used to transform source models into target ones (or vice-versa). We call *inter-modelling* to the activity of specifying how two or more modelling languages have to be related. Further examples of inter-modelling include specifications for model matching and traceability, inter-model consistency, and synchronization.

Frequently, the specifications of different inter-modelling activities (e.g. M2M transformation and model matching) are built separately from each other – even

if they relate *the same* modelling languages – and are written using different notations and tools. This produces scattered specifications that are prone to desynchronization and increase the maintenance effort. Moreover, all specifications that handle instances of the same meta-models need to be kept consistent, which is difficult to ensure if they lack formal semantics. Hence, a unified, formal notation able to specify different inter-modelling tasks would be very valuable.

Recently [6] we proposed a visual, declarative, bidirectional, formal language to describe M2M transformations. The language permits specifying allowed and forbidden relations between models of two modelling languages by means of patterns. Patterns have a formal semantics enabling checking whether two models are synchronized according to a pattern, and permitting static analysis. A synthesis procedure was developed in [6] to generate graph grammar rules solving two scenarios: source-to-target and target-to-source batch transformations.

In this paper we demonstrate that, in addition to transformation, our language can solve two further inter-modelling scenarios: model matching and model traceability. Hence, *the same* specification can solve different MDE tasks (transformation, matching and traceability) reducing the burden of developers. We also report on PAMOMO, an Eclipse tool that allows the definition of inter-modelling specifications, their analysis, and their operational use by compiling them into the Epsilon Object Language (EOL) [8]. The tool solves the following scenarios, for both model matching and model traceability:

1. Given two models M_1 and M_2, generate a trace model T relating both.
2. Given two models M_1 and M_2, and an existing trace model T,
   (a) verify whether T is valid (i.e. it has no missing or incorrect traces).
   (b) update T so that it becomes a valid trace model for M_1 and M_2.

The figure to the right shows the working scheme of our approach. The inter-modelling specification consists of a set of declarative triple patterns. This specification can be statically analysed (label 1) to check for conflicts between patterns, between patterns and the meta-models, and to assess meta-model coverage (i.e. check if all types are used by some pattern). Patterns can also be used operationally through their compilation into lower-level languages (label 2). In this paper, we compile them into EOL for model matching and model traceability, obtaining interoperability with EMF-based tools and an efficient implementation. Finally, patterns can be used for check-only scenarios (label 3) in order to find out whether two models are correctly traced and to detect incorrect or missing traces.

Paper organization. §2 presents our patterns for inter-modelling, which we use in §3 for model matching and traceability. §4 shows how to compile the patterns into OCL/EOL for these scenarios. §5 and §6 present tool support and a case study. We discuss related work in §7, and conclude in §8.
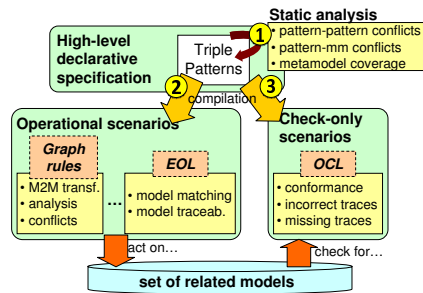
2 Our Pattern-based Inter-Modelling Language

In this section we briefly introduce our pattern-based language for inter-modelling. For technical details, the reader can consult [6].

Triple graphs. Our patterns are based on triple graphs [12], which are structures made of two graphs called source and target (S and T) related through a correspondence graph (C). Models can be represented as graphs with attributes in nodes and edges and with a type [4]. The correspondence graph is a graph in its own right, but we distinguish a special set of nodes M, called *mappings*. M is a subset of the set of nodes of the correspondence graph, $M \subseteq V^C$, and we define two functions $cs \colon M \to V^S$ and $ct \colon M \to V^T$ from M to the sets of nodes in the source and target graphs. These are called the correspondence functions, and are used to relate source and target nodes. Thus, we say that $x \in V^S$ is related to $y \in V^T$ iff $\exists m \in M$ s.t. $cs(m) = x$ and $ct(m) = y$. Altogether, a triple graph is a tuple $TrG = \langle S, C, T, M, cs, ct \rangle$.

We can relate two triple graphs through *triple morphisms*, e.g. when a triple graph represents a pattern that has to be found inside a bigger triple graph. A triple morphism $n \colon TrG_1 \to TrG_2$ is made of a triple of graph morphisms $n = \langle n^X \colon X_1 \to X_2 \rangle_{X \in \{S,C,T\}}$ relating the source, target and correspondence graphs of TrG_1 and TrG_2. In addition, the mappings of TrG_1 must be related to mappings of TrG_2, and the elements in TrG_1 that are not mappings cannot be identified to mappings of TrG_2 (i.e. $n^C(M_1) \subseteq M_2$ and $n^C(V_1^C \setminus M_1) \subseteq V_2^C \setminus M_2$).

Triple constraints. In order to interpret triple graphs as constraints, we substitute the set of data values in triple graphs by a finite set ν of sorted variables [6]. In this way, instead of concrete values, attributes point to variables of a given sort and their value can be constrained by a formula α. Altogether, a triple constraint is a tuple $CTrG = \langle TrG, \nu, \alpha \rangle$. It is important to note that a triple graph (i.e. two models related through a correspondence model) can be represented as a *ground* triple constraint where the formula α restricts the attributes to take exactly one value. Hence, we only need to consider triple constraints and not triple graphs anymore. As an example, the left of Fig. 1 shows a ground triple constraint taken from the class-to-relational example. The terms of the formula α are shown below, where the **and** connectives are omitted. Note that "=" denotes equality, not assignment.

Triple constraints can be related through *CTrG*-morphisms. A CTrG-morphism $a \colon CTrG_1 \to CTrG_2$ is made of a triple graph morphism with the following conditions: the formula α_2 of $CTrG_2$ must imply the formula α_1 of $CTrG_1$, and the same implication is demanded for the source and target restrictions of the formulae ($\alpha_2|_S \Rightarrow \alpha_1|_S$ and $\alpha_2|_T \Rightarrow \alpha_1|_T$). Roughly, the source $\alpha|_S$ (resp. target $\alpha|_T$) restriction of a formula α is the same formula but considering the variables of the source (resp. target) graph only [6]. In our current implementation, α can be any valid OCL expression given with EOL syntax.

Triple patterns. We use triple constraints as building blocks for triple patterns. A *triple pattern* describes in a declarative way a relation between two models. If the relation is allowed then we say the pattern is positive (P-pattern), whereas if the relation is forbidden then we say the pattern is negative (N-pattern). P-
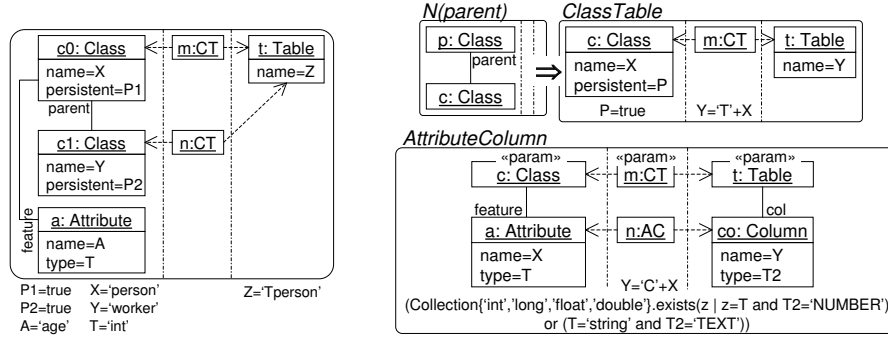
Fig. 1. Two related models as a ground triple constraint (left). Some P-patterns (right).

patterns are made of a constraint Q declaring the allowed relation, an optional positive pre-condition (or parameter) C with CTrG-morphism $q \colon C \to Q$, and a set $N_{Pre} = \{Q \xrightarrow{c_i} C_i\}_{i \in Pre}$ of negative pre-conditions (which may be empty). N-patterns consist of just one constraint Q forbidden to occur.

The right of Fig. 1 shows two P-patterns. The upper one has a negative pre-condition (`N(parent)`) and demands persistent, top-level classes to be related with tables. Its formula constrains the names of the related class and table. The lower P-pattern has as parameter all elements tagged with `<<param>>`, which are shown together with the main constraint Q. It states that attributes and columns should be related, but only if their owning class and table are related.

3   Model Matching and Model Traceability

Patterns are interpreted differently depending on the scenario. The M2M transformation scenario looks at patterns either source-to-target or target-to-source, checking whether patterns are source- or target-enabled [6]. For instance, in forward transformation, pattern `ClassTable` in Fig. 1 is to be interpreted as "given a class without parents, there must be a table". Instead, model matching and model traceability consider the source and target of patterns *at the same time*. Thus, in these cases, the same pattern is interpreted as "given a class without parents and a table with suitable name, there should be a trace relating them". Hence, given a pattern, we define a suitable *directed pre-condition* for the scenario at hand, which in model matching and traceability is called *trace pre-condition*. Next, we define the notion of pattern *enabledness*, which consists on finding an occurrence of the directed pre-condition that does not violate any negative pre-condition of the pattern. Finally, we build the notion of *satisfaction* for the particular scenario (here for model matching and traceability)[4].

Trace pre-condition. The *trace pre-condition* of a P-pattern is the constraint made of the source and target parts of the main constraint Q, together with the

---

[4] The formalization can be found at: `http://astreo.ii.uam.es/~jlara/PAMOMO.pdf`

parameter C and the formula. For example, the trace pre-condition of pattern `ClassTable` is made of objects `c`, `t` and *the complete formula*, while for pattern `AttributeColumn` it is made of objects `c`, `t`, `a`, `co`, `m` and the formula.

Trace enabledness. A P-pattern is *trace-enabled* if we find an occurrence of its trace pre-condition and none of its negative pre-conditions. In Fig. 1, `ClassTable` is enabled in the left constraint at objects $\{$`c0`, `t`$\}$, but not at objects $\{$`c1`, `t`$\}$ as the latter belongs to an occurrence of the negative pre-condition (i.e. `c1` has a parent). `AttributeColumn` is not trace-enabled because the table has no column.

Matched models. Two models are *matched* according to a specification, if each trace-enabled occurrence of every P-pattern in the specification belongs to an occurrence of the pattern's main constraint. This demands all suitable combinations of source and target elements to be traced. For N-patterns, we simply forbid their occurrence. The models to the left of Fig. 1 are correctly matched as the trace-enabled occurrence $\{$`c0`, `t`$\}$ of pattern `ClassTable` is included in an occurrence of the main constraint (i.e. a mapping `CT` exists).

Traced models. Two models are *traced* according to a specification if, for each trace-enabled occurrence of every P-pattern in the specification, the source part is traced with *some* (in contrast to *all*) suitable occurrence of the target, or the other way round. Thus, model traceability does not require all combinations of source and target elements to be traced. Here, the rationale for the trace model is that it could have been generated from a forward or a backward transformation, hence we also demand a "uniform" distribution of traces. This means that it is not allowed to have one occurrence of the source to be traced twice, whereas another occurrence that could have been related with the same target elements as the first one is not traced at all (and similarly for the target). As an example, the models in Fig. 1 are correctly traced.

Fig. 2 illustrates the difference between model matching and model traceability, through an example of two models having two classes and two tables, equally named. Model matching gives a unique minimal solution (left), whereas traceability gives two minimal solutions (right). Connecting the two classes to the same table is not a valid traceability solution, as there would be an unconnected table, but enough classes in the source to be connected with. Whereas the model matching solution cannot be generated by forward or backward transformation (it contains redundant traces), any of the traceability solutions can. In fact, the matching solution is the union of all traceability solutions. While two matched models are always correctly traced, the converse is not true in general.

4 Compilation of Patterns into OCL/EOL

We compile patterns into OCL/EOL to cover check-only and operational scenarios for model matching and traceability. In particular, we use OCL to check whether two models are correctly matched or traced, and EOL [8] to solve operational scenarios (i.e. to create a trace model from scratch so that two unrelated models become correctly matched or traced, and to recover the consistency of existing trace models by deleting incorrect traces and creating missing ones).
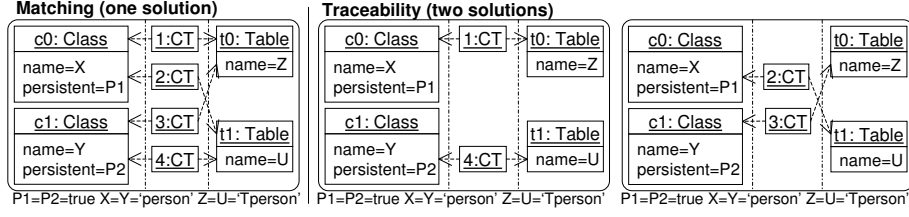
Matching (one solution) | **Traceability (two solutions)**

c0: Class	1:CT	t0: Table		c0: Class	1:CT	t0: Table		c0: Class		t0: Table
name=X persistent=P1	2:CT	name=Z		name=X persistent=P1		name=Z		name=X persistent=P1	2:CT	name=Z
c1: Class	3:CT	t1: Table		c1: Class		t1: Table		c1: Class	3:CT	t1: Table
name=Y persistent=P2	4:CT	name=U		name=Y persistent=P2	4:CT	name=U		name=Y persistent=P2		name=U

P1=P2=true X=Y='person' Z=U='Tperson' | P1=P2=true X=Y='person' Z=U='Tperson' | P1=P2=true X=Y='person' Z=U='Tperson'

Fig. 2. Model matching vs. model traceability.

4.1 Check-only scenario: Satisfaction of patterns by models

Here the aim is, given two models and their traces, to identify whether the models are matched or traced according to a pattern specification. Thus, the OCL code synthesized from the specification has to verify that each occurrence of the trace-precondition of every P-pattern actually satisfies the pattern, and that the models do not contain occurrences of the N-patterns. In this scenario, the only difference between matching and traceability is that the former demands universal existence of traces (i.e. for all combinations of source and target elements) whereas traceability demands them existentially (i.e. for at least one of them if they involve the same source or target elements).

Thus, for each P-pattern p, we generate an operation sat_p that (a) seeks all trace-enabled occurrences of p and (b) checks if they are related by traces as specified by p. For (a), the operation iterates on the nodes of the trace precondition and checks if: (i) all node's edges in the trace pre-condition can be mapped to links in the models; (ii) all mappings in the trace pre-condition can be mapped to traces in the models; (iii) the attribute conditions evaluate to *true* when symbols are replaced by concrete attribute values from the models; and (iv) there are no occurrences of the negative pre-conditions in the models. For (b), we try to extend each occurrence of the trace pre-condition found in (a) to the full main constraint. Next we show the compilation of a P-pattern for model matching using the OCL-like syntax of EOL, which e.g. uses keyword **operation** instead of **query**:

```
operation sat_p.name() : Boolean {
 return
 -- (a) for each occurrence of the trace-enabling conditions...
   patt_matching_forall⟨n₁⟩ implies...  ⎫
    patt_matching_forall⟨nᵢ⟩           ⎬  ∀nᵢ ∈ nodesᵖₚᵣₑ
     checkatt_p.name(n₁,...,nᵢ)         ⎭
 -- ... that does not violate any negative pre-condition,
     and not patt_matching_exists⟨mᵢ₁⟩ and...   ⎫
            patt_matching_exists⟨mᵢₖ⟩           ⎬  ∀Cᵢ ∈ Nᵖₚᵣₑ,
             checkatt_Cᵢ.name(n₁,...,nᵢ,mᵢ₁,...,mᵢₖ) ⎭  ∀mᵢₖ ∈ nodesᵖ_Cᵢ
 -- (b) check if it satisfies the main constraint
     implies
     patt_matching_exists⟨nᵢ₊₁⟩ and...  ⎫
      patt_matching_exists⟨nⱼ⟩           ⎬  ∀nⱼ ∈ nodesᵖₚₒₛₜ
```

```
        checkatt_p.name(n₁, ..., nᵢ, nᵢ₊₁, ..., nⱼ);
}
```

where `p.name` and C_i`.name` are the names of the pattern and its negative pre-condition C_i; `patt_matching_forall<nᵢ>` and `patt_matching_exists<nᵢ>` are replaced by expressions seeking all or one occurrence of node n_i satisfying (i-ii) and the ground terms in the formula assigning a concrete value to its attributes; `checkatt_X` are operations that evaluate the non-ground terms of the formula in X; $nodes^p_{pre} = \{n_i | n_i \in V^S_Q \cup V^T_Q \cup q(V^C_Q)\}$ are the nodes in the pattern trace pre-condition, where V^X_Y contains the graph X's nodes of Y (e.g. V^S_Q contains the nodes of the source graph S in the main constraint Q) and $q(V^C_Q)$ contains the correspondence nodes in Q which are parameter of the pattern; $nodes^p_{post} = \{n_j | n_j \in V^C_Q \setminus q(V^C_Q)\}$ are the traces created by the pattern; $N^p_{Pre} = \{Q \xrightarrow{c_i} C_i\}$ are the pattern negative pre-conditions; and $nodes^p_{C_i} = \{m_{ik} | m_{ik} \in (V^S_{C_i} \cup V^T_{C_i} \cup V^C_{C_i}) \setminus c_i(nodes^p_{pre})\}$ are the nodes in the negative pre-condition that are not in the trace pre-condition.

In the operation, `patt_matching_forall<nᵢ>` collects all nodes in the model with same type, edges and attribute values as node n_i in the pattern. To improve performance, these checkings are evaluated before entering an inner loop. Thus, the code that replaces `patt_matching_forall<nᵢ>` is the following:

```
nᵢ.type.allInstances().forAll(nᵢ |
   nᵢ.nav(nⱼ) = nⱼ and  }∀e ∈ edges^p_pre|src(e) = nᵢ, tar(e) = nⱼ, j ≤ i
   nₖ.nav(nᵢ) = nᵢ and  }∀e ∈ edges^p_pre|src(e) = nₖ, tar(e) = nᵢ, k ≤ i
   nᵢ.nav(nₗ) = nₗ and  }nᵢ ∈ V^M_C, ∀nₗ ∈ nodes^p_pre|cs(nᵢ) = nₗ or ct(nᵢ) = nₗ, l ≤ i
   nₘ.nav(nᵢ) = nᵢ and  }∀nₘ ∈ V^M_C|cs(nₘ) = nᵢ or ct(nₘ) = nᵢ, m ≤ i
   nᵢ.att = value and    }∀ condition v = value, where v stores attribute att of nᵢ
```

where `nᵢ.type` is replaced by n_i's type; $edges^p_{pre} = \{e | e \in E^S_Q \cup E^T_Q \cup q(E^C_Q)\}$ are the edges in the trace pre-condition; and $nav(n_j)$ in the expression n_i`.nav(nⱼ)` becomes the name of the association from node n_i to n_j. The generated pattern matching expressions are nested in operation `sat_p`, hence we implicitly order the nodes n_i. For efficiency we put first those nodes with higher number of links and ground constraints for their attributes. The code for `patt_matching_exists<nᵢ>` is similar but using *exists* instead of *forAll*.

After collecting the nodes, operation `checkatt_X` checks if they satisfy the non-ground part of the formula in X. The operation is generated for the trace-enabling condition of the pattern, its main constraint (both with same name but different parameters), and each negative pre-condition C_i. As an example, we show the operation generated for the trace-enabling condition:

```
operation checkatt_p.name(n₁ : n₁.type, ..., nᵢ : nᵢ.type) : Boolean {
   var v := nᵢ.att; } ∀ variable v storing an attribute of nᵢ ∈ nodes^p_pre
   return α_trace_precondition;
}
```

The operation `sat_p` for model traceability is similar, except that the expression that controls condition (b) (satisfaction of main constraint Q) just checks if the matched source elements satisfy Q with any combination of target elements, or if the target elements satisfy Q with any combination of the source ones.

Finally, from each N-pattern we generate one operation which checks the absence of occurrences of the N-pattern in the model. The operations are the same for model matching and traceability.

Example. Below we show part of the OCL code generated from `ClassTable` in Fig. 1, for the check-only model matching scenario:

```
operation sat_ClassTable() : Boolean {
  return  Class.allInstances().forAll(c | c.persistent=true implies
          Table.allInstances().forAll(t | checkatt_ClassTable(c, t)
  and not Class.allInstances().exists(p |
          c.parent.includes(p) and checkatt_ClassTable_parent(c, p))
  implies CT.allInstances().exists(m |
          m.source=c and m.target=t and checkatt_ClassTable(c, t, m))));
}
operation checkatt_ClassTable( c:Class, t:Table ) : Boolean {
  var X:=c.name;  var Y:=t.name; return Y='T'+X;
}
```

4.2   First operational scenario: Creation of correct traces

In operational scenarios we are given two models which can be related or not, and the aim is creating missing traces and deleting incorrect ones. For the former, from each P-pattern p we generate an EOL operation `rule_p` that (a) looks for a trace-enabled occurrence of the pattern and (b) applies the pattern to it (i.e. it creates the traces according to the pattern). Trace-enabledness is checked as in the check-only scenario, but includes two additional conditions: (v) the pattern must not have been applied to the same objects before (termination condition), and (vi) the result of applying the pattern must not violate any N-pattern in the specification.

Termination condition (v). In model matching we must ensure that the elements in the trace-enabled occurrence of a pattern are not related as specified by the pattern. For this purpose we generate an extra condition which is equal to that generated in the check-only scenario to check satisfaction of the main constraint (three last lines in the body of operation $sat\_p.name$), but preceded by *not* instead of *implies*. This avoids enforcing a pattern twice for the same objects. In traceability we generate two stronger conditions checking that the source structure is not related to some occurrence of the target one, and vice-versa.

N-patterns (vi). In order to ensure that applying a P-pattern does not create occurrences of N-patterns, we encapsulate the creation actions into transactions which are rolled back if their execution results in an N-pattern violation. For efficiency reasons, after applying a P-pattern only those N-patterns which include elements created by the P-pattern are checked.

Creation of traces. If a set of objects satisfy all trace-enabling conditions, then they are passed as parameters to an operation `apply_p` which creates the nodes, traces and edges that appear in the correspondence graph of Q, but not in the positive pre-condition.

$$\boxed{\begin{array}{l}
\textbf{operation } apply\_\text{p.name}(n_1 : \texttt{n}_1.\texttt{type}, \ldots, n_i : \texttt{n}_i.\texttt{type}) : \textbf{Boolean } \{ \\
\quad \text{-- \textit{creation of new nodes}} \\
\quad var\ \texttt{v.id} \ : \ new\ \texttt{v.type}; \quad \} \ \forall v \in nodes^p_{post} \\
\quad \text{-- \textit{creation of new edges}} \\
\quad n_k.\texttt{nav}(\texttt{n}_\texttt{l}) := n_l; \quad \} \ \forall e \in edges^p_{post},\ with\ src(e) = n_k,\ tar(e) = n_l \\
\quad \text{-- \textit{creation of new correspondence functions}} \\
\quad var\ n_c.\texttt{nav}(\texttt{n}_\texttt{s}) := n_s; \quad \Big\} \ \ \forall n_c \in V^M_Q \cap nodes^p_{post},\ with\ n_s \in V^S_Q, \\
\quad var\ n_c.\texttt{nav}(\texttt{n}_\texttt{t}) := n_t; \quad \ \ n_t \in V^T_Q,\ cs(n_c) = n_s,\ ct(n_c) = n_t \\
\quad \textbf{return true}; \\
\}
\end{array}}$$

where $\texttt{v.id}$ is replaced by a unique identifier for node v.
Example. Part of the generated matching code for pattern `ClassTable` is:

```
operation rule_ClassTable() : Boolean {
  return  Class.allInstances().exists(c | c.persistent=true and
          Table.allInstances().exists(t | checkatt_ClassTable(c, t)
  and not Class.allInstances().exists(p |
          c.parent.includes(p) and checkatt_ClassTable_parent(c, p))
  and not CT.allInstances().exists(m |
          m.source=c and m.target=t and checkatt_ClassTable(c, t, m))
  and apply_ClassTable(c, t))); }
operation apply_ClassTable( c:Class, t:Table ) : Boolean {
  var m:new CT;  m.source:=c;  m.target:=t;  return true; }
```

4.3   Second operational scenario: Deletion of incorrect traces

The previous operational mechanism ensures that the needed traces exist, but does not guarantee the absence of incorrect traces. This is so because it iterates on occurrences of the source and target nodes creating valid traces, but does not iterate on the occurrences of traces checking their correctness. Hence, two related models may have incorrect traces (apart from the correct ones) if somebody manually added an incorrect trace between them, or if the models evolved so that some traces became incorrect. Here we make a closed world assumption: only those traces that are correct according to the specification should exist.

   In order to achieve this, we generate additional EOL operations that detect and delete incorrect traces. The operations check that, whenever there is a trace in the correspondence model, it is because some P-pattern demands its presence and it does not belong to an occurrence of any N-pattern.

   We generate two types of operations, enforcing two levels of trace correctness. The first operation type is called *relaxed* and it does not take into account the negative pre-condition of patterns, since a pattern with negative pre-conditions specifies what should happen if the negative pre-conditions are not found but not if they are found. However the synthesized EOL code for trace creation does not enforce a pattern if its negative pre-conditions are found; therefore the second operation type checks that only those traces that our previous compilation is able to create actually exist. This second operation type is called *strict*. For space constraints we only show the compilation of the first operation.

$$
\boxed{
\begin{array}{l}
\textbf{operation } relaxed\_1\_\texttt{t.type} \ (t : \texttt{t.type}) \ \{ \\
\quad \textbf{if } (\textbf{not } \texttt{patt\_matching\_exists}\langle \texttt{n}_1 \rangle \textbf{ and}\dots \\
\qquad\qquad \texttt{patt\_matching\_exists}\langle \texttt{n}_i \rangle \\
\qquad\qquad\quad checkatt\_\texttt{p.name}(n_1, \dots, n_i) \dots) \\
\qquad\quad \{ \ \texttt{t.type}.allInstances().remove(t); \ \text{-- } remove \ correspondence \ object \\
\qquad\qquad \textbf{delete } t; \ \} \\
\}
\end{array}
}
\quad
\begin{array}{l}
\forall p \in enabling_t, \\
\forall n_i \in nodes^p_{pre} \cup nodes^p_{post} \setminus t
\end{array}
$$

where $enabling_t = \{p | p \ is \ a \ P\text{-}pattern, \exists n \in nodes^p_{post} \ with \ n.type = t\}$ is the set of P-patterns in the specification that create traces of type t.

Example. The first type of relaxed operation generated for trace `CT` is:

```
operation relaxed_1_CT( mt:CT ) {
  if (not Class.allInstances().exists(c |
          c.persistent=true and mt.source=c and
          Table.allInstances().exists(t |
          mt.target=t and checkatt_ClassTable(c, t, mt))))
      { CT.allInstances().remove(mt); delete mt; }
}
```

The generated EOL code for the operational scenarios works incrementally. Thus, given source and target models connected through an arbitrary trace model, the program invokes the deleting operations to delete the incorrect traces, and then the creation ones to reestablish trace correctness.

5   Tool Support

We have developed an Eclipse tool, called PAMOMO (`http://astreo.ii.uam.es/ ~jlara/pamomo/main.htm`), to build pattern specifications. It supports two modes of execution: *off-line* and *on-line*. In the former the designer can validate a specification or generate different files with EOL code to perform model matching, traceability, relaxed/strict deletion, or evaluate if a specification is satisfied by models. In this execution mode the specification is compiled once and the result can be used afterwards for any incoming models, or be integrated in other tools and model driven tasks. In the *on-line* mode the designer selects the incoming source, correspondence and target models and the specification is applied to them for the chosen scenario. A ModeLink [9] file is generated showing the result in an Eclipse three-pane window, the one in the middle containing the generated trace model (see e.g. Fig. 6). The user can manipulate the result in order to e.g. annotate traces with additional information.

PAMOMO also supports analysis of meta-model coverage, identifying which types are included in each positive/negative pre-condition, main constraint or N-pattern. This has different interpretations depending on the scenario. For example, a P-pattern that defines as parameter a trace type that is not created by any other P-pattern in the specification may be useless.

Fig. 3 shows the PAMOMO meta-model used to define pattern specifications. It shows that *Specification*s are made of positive and negative patterns, both sub-classes of *Pattern*. Patterns have a main constraint (role *constraint*), an optional

positive pre-condition, and a set of negative pre-conditions, all modelled through class *ConstraintTripleGraph*. This class is made of three graphs with roles *source*, *target* and *correspondence*. The corrrespondence graph is a special kind of graph which may contain mappings that point to source and target objects.
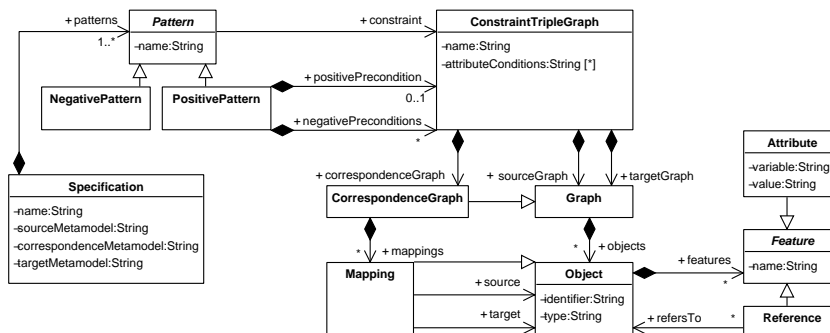

Fig. 3. Meta-model of Pamomo.

On top of this meta-model we have built a textual concrete syntax editor for Pamomo with XText (`http://eclipse.org/Xtext`). This editor takes a textual representation of a specification like the one shown to the right of Fig. 5, and parses it to our model-based internal representation. Then, the code generators we have built synthesize EOL files for the chosen scenario, following the algorithms of previous section.

6 Example

In the literature, model matching has been mainly used to compare instances of the same meta-model. Here we show that it can be used for very different purposes, in particular to implement a GoF design pattern [5] discovery mechanism. On the one hand we have Ecore models where we want to identify instances of design patterns, and on the other hand a pattern design vocabulary with the definition of different design patterns and the roles participating in them. Fig. 4 shows part of the meta-model triple for this situation, which in the real case contains the complete Ecore meta-model to the left, and additional role specializations (apart from those for classes, operations and references) to the right. The correspondence meta-model binds roles to UML elements and groups the bindings of each pattern instance through class *Instance*.

The meta-model permits annotating Ecore models with design pattern roles. Besides, we define a Pamomo specification to automate the identification of design patterns in the Ecore models and annotate their elements with the roles they play in the design patterns. For instance, Fig. 5 shows the Pamomo pattern for the *Proxy* design pattern. The pattern identifies occurrences of the proxy, and
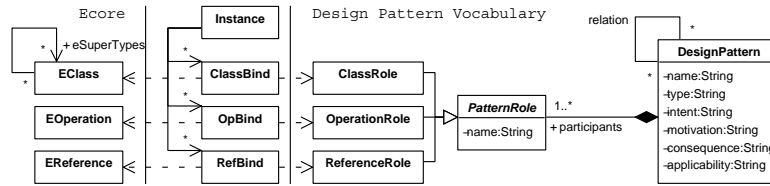
Fig. 4. Meta-model to annotate Ecore models with roles in a design pattern vocabulary.

requires that the operations in *Subject*, *RealSubject* and *Proxy* have the same name, modelled with variables $n1$, $n2$ and $n3$, all having the same value (see condition). The pattern may define additional conditions, e.g. that the proxy defines one public operation for each public operation in the subject, and it does not define further public operations apart from these. We could also define another pattern to annotate all operations in a proxy instance, hence allowing variability on the number of operations that the *Proxy* wraps.
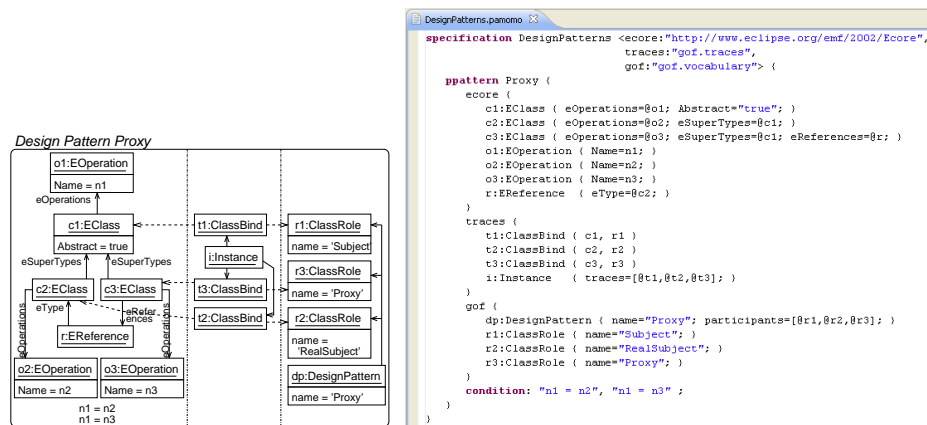

Fig. 5. Specification of the *Proxy* design pattern.

With our approach we formalize the structure of design patterns as an inter-modelling specification. If we apply this specification to an EMF model and a design pattern vocabulary model (instances of the meta-models in Fig. 4), we can identify instances of the patterns in the Ecore models, by using the mechanism for creation of traces in model matching. Fig. 6 shows the result provided by our tool in a simple example. The process identified one instance of the proxy in the model to the left. By selecting the created traces in the middle we can see the particular role assigned to each element in the EMF model. In the figure, the first trace binds role *Subject* to class *Graphic* in a proxy instance.
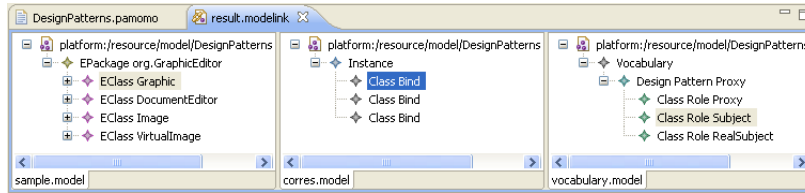
Fig. 6. Model matching result: EMF model, trace model, design pattern vocabulary.

The mechanism for trace manipulation is incremental: if we modify the EMF model after having identified design pattern instances, we can apply our operational mechanisms for deletion of incorrect traces (in case some instance was destroyed), as well as to identify new instances of patterns. Moreover, we plan to use PAMOMO for Ecore model completion w.r.t. design patterns by allowing users to manually annotate objects in the EMF model (i.e. assign them a role in the pattern vocabulary). We could then apply a backward transformation to create Ecore objects to obtain a correct instance of the design pattern.

7   Comparison with Related Work

Our long-term goal is providing a formal yet practical approach to integrate inter-modelling tasks. Whereas in [11] the focus is on representing sets of related models through macromodels (theoretically based on institutions), we provide a declarative, bidirectional language to describe inter-model relations, as well as a tool to enforce such relations. The goal in [2] is developing model management operators for schema mapping and data integration, while in [13] the authors use mega-models to distinguish between high- and low-level traceability models.

Among the existing traceability approaches, the Atlas Model Weaver (AMW) [1] supports the creation of weaving models (similar to our correspondence meta-model) establishing links between meta-model elements. This makes AMW usable only when the source and target meta-models are very similar, and just derives straight-forward source-to-target transformations. The specification of complex conditions enabling the creation of traces, like e.g. the one in Fig. 5, requires in addition specifying conditions *at the model level* by means of patterns of source and target instances, not supported in AMW. The work in [3] is based on a traceability meta-model in which OCL-like consistency conditions can be given. Note that both approaches are specific to traceability and are not formally founded, and therefore cannot be analysed. Finally, QVT-Relations [10] (QVT-R) can be used for model traceability by setting all domains as check-only. However, even in check-only mode, specifications have a direction and relations have to be interpreted either source-to-target or target-to-source.

Regarding model matching, existing approaches permit comparing models expressed in the same language, typically UML [14], and the customization of the comparisons is usually limited. However the advent of Domain Specific Languages makes evident the need for comparing heterogeneous models. In this

respect, ECL is a dedicated language for model comparison [7] which supports heterogeneous models. However its rules are restricted to compare one source element with one target element, hence expressing a pattern like the one in Fig. 5 would require coding by hand the pattern-matching code that we generate automatically. Moreover, to the best of our knowledge, no model matching approach provides a formal foundation enabling the analysis of specifications. Regarding limitations, the advantages of formality with respect to analysis capabilities come to the price of less expressiveness than other low-level operational languages [7, 8] (e.g. we do not provide primitives for creating elements in arbitrary loops).

Although TGGs can be used for model matching [12], their compilation into operational rules does not produce application conditions, and hence extra control mechanisms have to be designed ad-hoc. Using TGGs for check-only scenarios would require model parsing, and lacks an equivalent to our N-patterns. Regarding QVT-R, its semantics is not suitable for model matching because it is not possible to consider all domains at the same time, as our concept of trace-enabledness does. Moreover, the lack of an explicit concept of *trace* makes difficult its use for model matching.

Table 6 summarises the comparison of PAMOMO with the mentioned approaches. The symbols $\sqrt{}$ and $-$ indicate whether they support a given feature or not. The table shows whether the approaches can be used for traceability or matching (columns 2 and 3), if they admit an explicit trace meta-model (column 4), have a formal foundation (column 5), have a declarative style (column 6), admit non-constructive primitives similar to our N-patterns (column 7), whether the traces are defined at the meta-model level or if it is possible to define additional constraints at the model level as rules or patterns (mm vs. m, column 8), and if they permit relating heterogeneous languages (column 9). As it is apparent, PAMOMO is the only approach that supports both matching and traceability, under a unified formal semantics, making it suitable for inter-modelling tasks.

Table 6. Comparison of different approaches for model traceability and matching.

	Traceab.	Matching	Trace MM	Formal	Declarative	Non-constructive	m*	Heterogeneous
PAMOMO	$\sqrt{}$	$\sqrt{}$	$\sqrt{}$	$\sqrt{}$	$\sqrt{}$	$\sqrt{}$	m	$\sqrt{}$
AMW	$-$	$-$	$\sqrt{}$	$-$	$\sqrt{}$	$-$	mm	$\sqrt{}$
ECL	$-$	$\sqrt{}$	$-$	$-$	$-$	$-$	m	$\sqrt{}$
UMLDiff	$-$	$\sqrt{}$	$-$	$-$	$-$	$-$	$-$	$-$
QVT-R	$\sqrt{}$	$-$	$-$	$-$	$\sqrt{}$	$-$	m	$\sqrt{}$
TGGs	$-$	$\sqrt{}$	$\sqrt{}$	$\sqrt{}$	$\sqrt{}$	$-$	m	$\sqrt{}$

8 Conclusions and Future Work

This paper has shown the use of our pattern-based approach to specify model matching and model traceability conditions. For these scenarios, patterns can be used in check-only mode to test satisfiability, and in operational (incremental) mode to manipulate the trace model. We have shown realizations of these

two activities using OCL and EOL respectively. Our patterns provide a unified, formal approach to inter-modelling, as pattern specifications can also be used to solve M2M transformation scenarios [6]. We have also introduced PAMOMO, an EMF-based tool that allows editing pattern-specifications, their static analysis, and their compilation into EOL for model matching and traceability.

On the practical side, we are working on optimizing the pattern matching algorithms, as well as in extending PAMOMO to solve M2M transformation scenarios. For this purpose we need to combine EOL with constraint solving techniques. On the theoretical side, we are currently working on new analysis techniques and on extending the expressivity of patterns.

References

1. AMW. ATLAS Model Weaver. http://wiki.eclipse.org/AMW.
2. P. A. Bernstein and S. Melnik. Model management 2.0: manipulating richer mappings. In *SIGMOD*, pages 1–12. ACM, 2007.
3. N. Drivalos, D. Kolovos, R. Paige, and K. Fernandes. Engineering a DSL for software traceability. In *SLE08*, volume 5452 of *LNCS*, pages 151–167. Springer, 2008.
4. H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of algebraic graph transformation*. Springer-Verlag, 2006.
5. E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.
6. E. Guerra, J. de Lara, and F. Orejas. Pattern-based model-to-model transformation: Handling attribute conditions. In *ICMT'09*, volume 5563 of *LNCS*, pages 83–99, 2009.
7. D. S. Kolovos. Establishing correspondences between models with the Epsilon Comparison Language. In *ECMDA-FA'09*, volume 5562 of *LNCS*, pages 146–157. Springer, 2009.
8. D. S. Kolovos, R. F. Paige, and F. Polack. The Epsilon Object Language (EOL). In *ECMDA-FA'06*, volume 4066 of *LNCS*, pages 128–142. Springer, 2006.
9. Modelink. http://www.eclipse.org/gmt/epsilon/doc/modelink/.
10. QVT. http://www.omg.org/docs/ptc/05-11-01.pdf.
11. R. Salay, J. Mylopoulos, and S. Easterbrook. Using macromodels to manage collections of related models. In *CAiSE'09*, volume 5565 of *LNCS*, pages 141–155. Springer, 2009.
12. A. Schürr. Specification of graph translators with triple graph grammars. In *WG'94*, volume 903 of *LNCS*, pages 151–163. Springer, 1994.
13. A. Seibel, S. Neumann, and H. Giese. Dynamic hierarchical mega models: comprehensive traceability and its efficient maintenance. *SOSYM*, in press, 2010.
14. Z. Xing and E. Stroulia. UMLDiff: an algorithm for object oriented design differencing. In *ASE'05*, pages 54–65. ACM, 2005.