Electronic Communications of the EASST
Volume 47 (2012)

Proceedings of the 11th International Workshop on Graph Transformation and Visual Modeling Techniques (GTVMT 2012)

Inter-Modelling with Graphical Constraints: Foundations and Applications

Juan de Lara, Esther Guerra

16 pages

Guest Editors: Andrew Fish, Leen Lambers
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Inter-Modelling with Graphical Constraints: Foundations and Applications

Juan de Lara[1], **Esther Guerra**[1]

[1] (Juan.deLara, Esther.Guerra)@uam.es
Department of Computer Science
Universidad Autónoma de Madrid, Spain

Abstract: Model-Driven Engineering (MDE) promotes an active use of models in the different phases of the development, so that the construction of systems usually involves a number of models expressed in different languages and levels of abstraction; therefore, there is the constant need to compare, generate and update models and their relations.

We call *inter-modelling* to the activity of building models that describe how modelling languages should be related. This includes many MDE activities like the specification of model-to-model transformations, the definition of model matching and traceability constraints, and the development of inter-model consistency mantainers. While most approaches build different operational programs to handle each activity separately, we propose using a high-level specification language called PAMOMO to specify inter-models in a declarative, graphical, bidirectional way. This specification can be compiled into operational mechanisms to solve different inter-modelling activities like transformation, model comparison and traceability support. Other usage scenarios for PAMOMO are the specification of transformation contracts and the automated testing of transformations.

Keywords: Model Transformation, Inter-Modelling, Graph Constraints, Bidirectionality

1 Introduction

Models are the core assets of the development in Model-Driven Engineering (MDE). The rationale is that they allow a high-level, more intentional description of systems, with less accidental details because they frequently use concepts of the problem domain and not of the solution space. Hence, models are used to specify, test, maintain and generate code for the final applications. In this context, model manipulations become a key activity which may involve one model (e.g. for model animation, simulation and refactoring) or several models (e.g. when transforming a model to a different modelling language, or when comparing, merging or establishing traceability links between models). In the latter case, the usual approach is building a different program for each particular manipulation, regardless they involve the same modelling languages. This sometimes leads to an unnecessary proliferation of heterogeneous programs which become difficult to maintain synchronized when the involved languages change.

We call *inter-modelling* [GLO11, GLKP10a] to the activity of building models that describe

how the models of different modelling languages should be related. Most manipulations that involve several models can be specified using *inter-models*. In this paper, we give an overview of our inter-modelling language PAMOMO (for Pattern-based Model-to-Model specification language) [GLKP10b]. This is a bi-directional, declarative *specification* language (in contrast to an implementation language) where the same specification can be used to solve several model manipulation scenarios. For this purpose, we provide several notions of conformance or satisfaction, and several compilations into operational mechanisms. This paper will present the foundations of PAMOMO [GLKP10b, GLO11, LG08] as well as some applications [GLKP10a, GLW+12, GL12].

The rest of the paper is organized as follows. Section 2 introduces some inter-modelling activities. Then, Section 3 overviews the foundations of PAMOMO, including its syntax, different notions of conformance, and compilations into operational mechanisms for different scenarios. Section 4 illustrates its use with two examples. The first one applies an inter-modelling specification to the discovery of design patterns in meta-models. The second one uses PAMOMO as a means to specify contracts for model transformations, and their application for automated testing. Section 5 compares with related work and Section 6 concludes the paper.

2 Inter-Modelling Activities

This section introduces some inter-modelling activities of interest, namely, model-to-model transformation, model matching and traceability between models.

In model-to-model transformation, a source model conforming to a source meta-model is transformed into a target model conforming to a target meta-model. In the simplest case, the target model is created from scratch. This scenario, called *batch transformation*, is depicted schematically in Figure 1(a). In this case, the transformation creates a target model together with a trace model containing mappings between source and target model elements. If we need to transform also backwards (from target to source), and we are using an operational transformation language, then we need to implement a different transformation for each direction. Instead, we propose using bi-directional, declarative inter-modelling specifications from which deriving operational transformations for both directions. Thus, a model transformation definition can be seen as an inter-modelling specification that states the conditions under which a target model is considered a correct transformation of a source model, and vice versa. For each direction we use a different notion of conformance. The figure uses $\texttt{SAT}_F$ to convey that we use the specification to check *forward conformance*: whether the target model can be considered a correct transformation of the source. Backward conformance, denoted as $\texttt{SAT}_B$, is defined symmetrically. We say that two models are *synchronized* if they conform both forwards and backwards to the specification.

A more complex scenario is the so-called *incremental transformation*. In this scenario, a source model is initially transformed into a target model (e.g. using a batch transformation), then the source model is updated, and the incremental transformation must propagate the performed changes to the target model. This is usually done for efficiency reasons, to avoid having to regenerate the target model from scratch, or to prevent the overriding of any manual change made to the target model. This scenario is illustrated in Figure 1(b). Finally, in the more complex

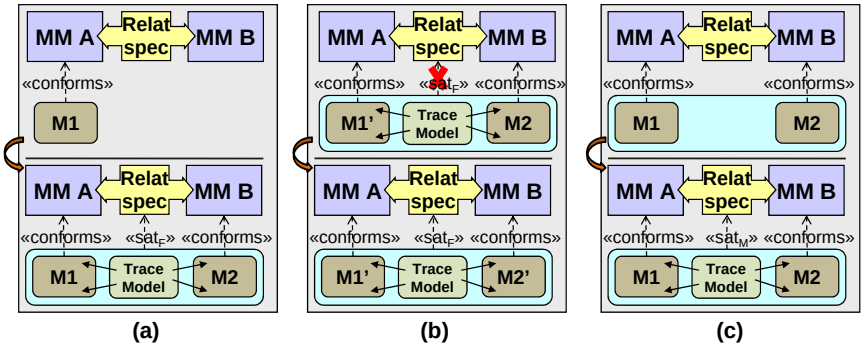


Figure 1: Some inter-modelling activities: (a) forward batch transformation, (b) forward incremental transformation, (c) model-matching.

synchronization scenario, both the source and target models may be modified after they were consistent, and the operational mechanism may have to update them to recover their consistency.

In addition to model transformations, another interesting inter-modelling activity is to establish *traceability* between models [PDK+11, WP10]. In this case, given two unrelated models, the operational mechanism must produce a trace model relating source and target elements so that both models (or parts of them) become synchronized. This can be used to trace the origin or provide semantics to the different model elements, or as a previous step to the synchronization scenario. In the latter case we first produce the traces between two given unrelated models, and then apply the synchronization operational mechanism.

Model matching is the last inter-modelling activity we tackle. It is useful when we want to compare models, or as a previous step to model merging [Kol09]. The scenario is depicted in Figure 1(c). In this case, a trace model is created identifying all elements in the source and target models considered equivalent according to the inter-modelling specification. After the operational mechanism has created the trace model, the related models should conform to the specification. This can be checked by using a special notion of conformance, different from the previous ones, depicted in the figure as $\mathtt{SAT}_M$.

Altogether, an inter-modelling specification is used in two ways. First, it is used to generate operational mechanisms for solving a particular scenario (e.g. creating a target model given a source one, as shown in Figure 1(a)). Secondly, it is used to *check* the conformance of a set of related models for a particular scenario. The next section provides an overview of our specification language for inter-modelling, the way it is used to check conformance of models with respect to specifications, and the methods to synthesize operational mechanisms for the different scenarios.

3 PAMOMO: A Specification Language for Inter-Modelling

PAMOMO is a pattern-based, formal, declarative, bi-directional language. In our approach, shown in Figure 2, designers can use this language to build inter-modelling specifications stating how the models of two modelling languages should be related (label "1" in the figure). Patterns in a specification can be positive or negative, to express allowed or forbidden relations between

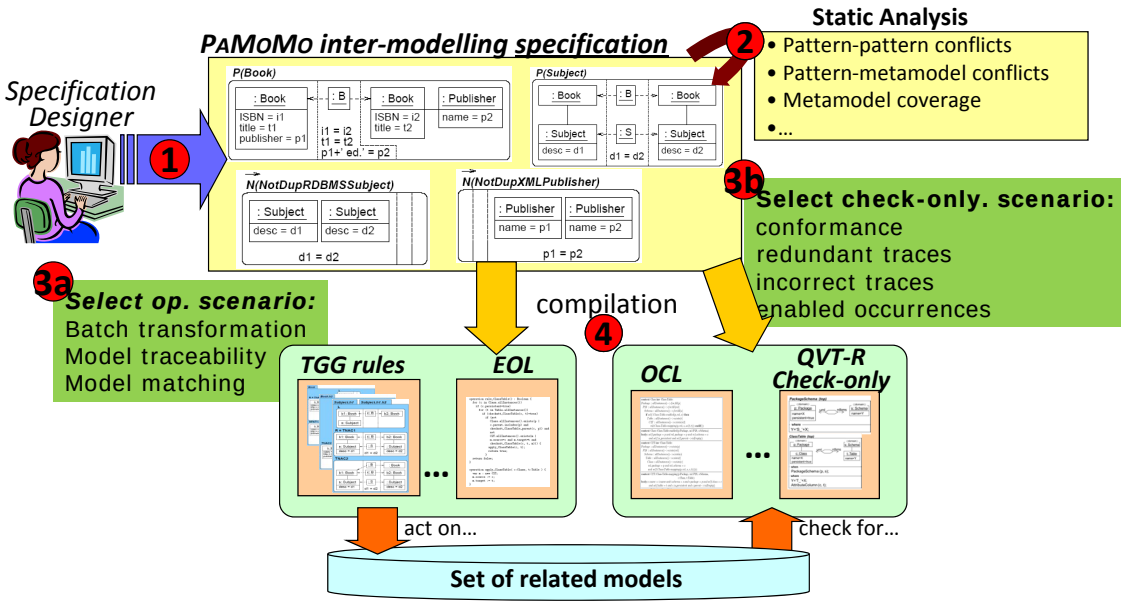


Figure 2: Architecture of our approach.

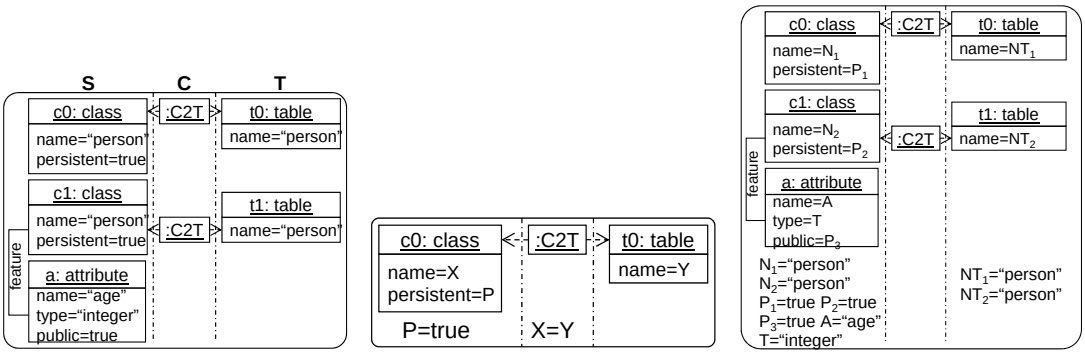
two models. Such a specification can be analysed (label "2" in the figure), for example to detect conflicts or contradictions between patterns, to find redundant patterns, or to identify disconformities between the patterns and the meta-models of the languages being related. It is also possible to measure the degree of coverage of a (source or target) language by a specification, and analyse whether certain classes only participate in negative (i.e. non-generative) patterns. This provides an indication of the completeness of a specification.

Specifications can be used both in operational and check-only modes for each particular inter-modelling scenario (up to now, batch transformation, model traceability and model matching). For the operational mode, we generate operational Triple Graph Grammar rules [Sch94] or programs written in the Epsilon Object Language (EOL) [KPP06] that implement the expected behaviour for the chosen scenario. EOL is a variant of OCL [OCL] with side effects that enable for example creating new objects or setting attribute values. For the check-only mode, we generate OCL code that can be used to assert conformance of models with respect to the specification for the chosen scenario, as well as to identify incorrect, redundant or missing traces between models. Additionally, for the batch transformation scenario, we can generate QVT-Relations code [QVT] that can be executed in check-only mode with a QVT engine like ModelMorf [Mod] to check for disconformities of models with respect to the specification. As we will see in Section 4.2, this is especially useful in model transformation testing [GLW+12].

3.1 Models and their relations, algebraically

The building blocks of PAMOMO patterns are the so-called *constraint triple graphs*, an algebraic construction that we have defined to relate two models. In this section we present this construct. We keep the discussion on an informal level, and refer to [GLO11] for technical details.

We represent models as attributed typed graphs [EEPT06]. In this kind of graphs, attribute



(a) A triple graph (b) A CT-graph (c) A CT-graph representation of the triple graph in (a)

Figure 3: A unified view of triple graphs and constraints.

values are represented as data nodes, and attributes are edges connecting graph nodes with data nodes. For inter-modelling, we need to relate models through a trace model. For this purpose, we use triple graphs [Sch94], which are structures made of a source and a target graph (S and T) related through a correspondence graph (C). The correspondence graph represents the trace model and contains the mappings (or traces) between the other two graphs. These mappings are given by two graph morphisms $cs\colon C \to S$ and $ct\colon C \to T$. A graph morphism is a collection of set functions relating the sets of nodes, edges and attributes in two graphs, and preserving the graph structure. As an example, Figure 3(a) shows a triple graph relating a class diagram (in abstract syntax) and a relational data-base schema. The source graph contains one edge (labelled *feature*) and three *graph* nodes: *c0*, *c1* and *a*. The first two graph nodes have two attributes each (*name* and *persistent*), while the last graph node has three attributes. We represent graphs using the usual UML object diagram notation, showing attributes inside a box compartment close to the owning graph node, and indicating the typing after the node name, separated by a colon.

An inter-modelling language needs to express constraints on related models. For this purpose, we define *constraint triple graphs* (CT-graphs in short) as triple graphs where the data nodes (i.e. the attribute values) are replaced by a finite set of variables and a formula α constraining their value. Additionally, CT-graphs are defined over a signature $\Sigma = (S, op)$ that contains the sorts and operations used by the formula α. As an example, Figure 3(b) shows a CT-graph. We usually omit the conjunctions in the formula α and split it in three parts: α^S, containing the terms dealing with variables of the source model only, α^T containing the terms dealing with variables of the target model only, and α^C with the terms that relate variables in both models. We display α^S inside the source compartment (see `P=true` in the figure), α^C in the middle (see `X=Y` in the figure), and α^T in the target compartment (`true` in this case as no formula is shown).

Triple graphs can be represented as CT-graphs by creating a variable for each attribute value, and including a conjunctive term with the equality of each variable and its value in the formula, as Figure 3(c) shows. In this way, we only need to handle one kind of structure. Thus, finding an occurrence of a CT-graph in a triple graph amounts to defining a morphism between two CT-graphs, called CT-morphism. A CT-morphism $f\colon M_1 \to M_2$ between two CT-graphs M_1 and M_2 is made of a structural morphism (i.e. a morphism on triple graphs), with the condition that

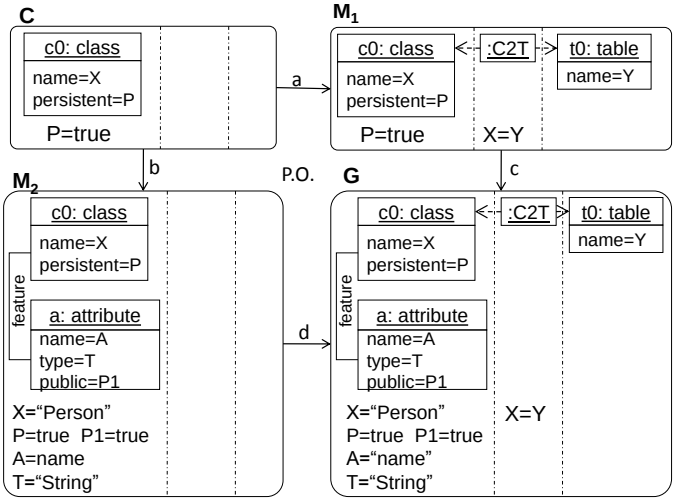


Figure 4: Glueing of CT-graphs: Pushouts.

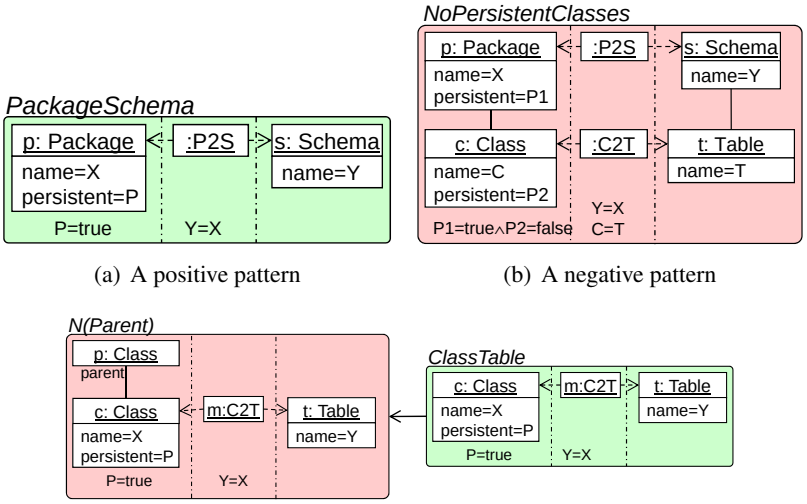
the formula α_1 in M_1 should be weaker or equal than the formula α_2 in M_2. More in detail, CT-morphisms must fulfil three implications: $\alpha_2^S \Rightarrow \alpha_1^S$, $\alpha_2^T \Rightarrow \alpha_1^T$, and $\alpha_2 \Rightarrow \alpha_1$. For instance, there are two CT-morphisms from the CT-graph in Figure 3(b) to the one in Figure 3(c). The first one identifies equally named elements, and the second one identifies $c0$ with $c1$ and $t0$ with $t1$.

A useful operation in our context is the merging or glueing of two CT-graphs M_1 and M_2 through some common elements given by a CT-graph C and two CT-morphisms $a\colon C \to M_1$ and $b\colon C \to M_2$. This is called a *pushout* in category theory [EEPT06]. This operation yields a new CT-graph G and two CT-morphisms from M_1 and M_2 to G. The resulting CT-graph G contains a copy of the elements present in the other two graphs, without duplication, and its formula is the conjunction of the formulas in M_1 and M_2. Figure 4 shows an example of pushout, where CT-graphs M_1 and M_2 are glued through their common element *c0*. The resulting CT-graph G contains the different elements of M_1 and M_2 glued via the common part, so that element $c0$ only appears once, and its formula is the conjunction of the formulas in M_1 and M_2.

3.2 The PAMOMO specification language

In this section we briefly introduce PAMOMO, our pattern-based language for inter-modelling. For technical details, the reader can consult [GLO11].

A PAMOMO specification is made of a set of patterns. Patterns are built on the notion of CT-graph presented in the previous subsection. A pattern describes in a declarative way a relation between two models. If the relation is allowed, then we say that the pattern is positive (P-pattern), whereas if the relation is forbidden, then we say that the pattern is negative (N-pattern). P-patterns are made of a main CT-graph Q defining the allowed relation, an optional *enabling condition* C with a CT-morphism $q\colon C \to Q$, and a set $NPre = \{c_i\colon Q \to C_i\}_{i \in Pre}$ of *disabling conditions*, which may be empty. The enabling and disabling conditions can be used to reduce the scope of a pattern to the locations where the enabling condition is met, and the disabling conditions are not. N-patterns have the same structure as P-patterns, but the interpretation is that the relation defined by the CT-graph Q is forbidden to occur.



(a) A positive pattern

(b) A negative pattern

(c) A pattern with a disabling condition

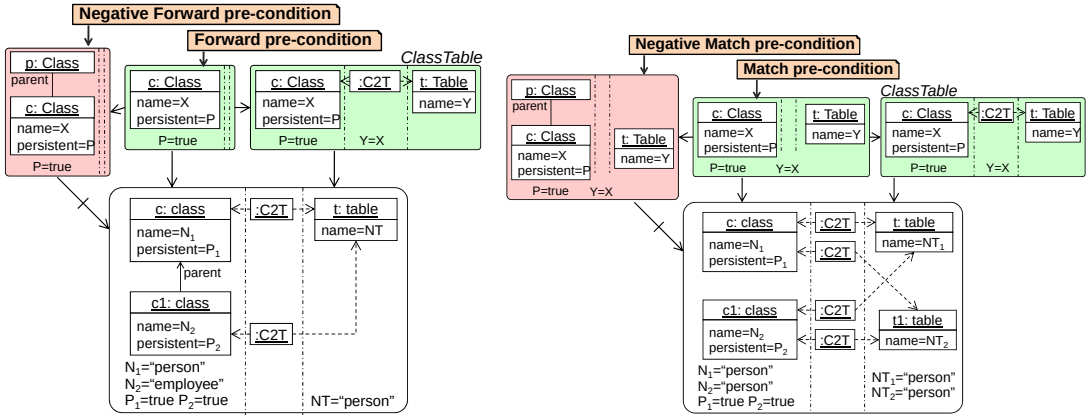
Figure 5: PAMOMO patterns.

As an example, Figure 5 gathers some patterns that belong to the specification of the "classical" class-to-relational transformation (see the appendix of [QVT]). Figure 5(a) shows a P-pattern specifying how `Packages` and `Schemas` should be related. In particular, it specifies that if there is a trace mapping between a package and a schema, then they have the same name and the package is persistent. We will see that the precise meaning of the pattern depends on the inter-modelling scenario to be solved. Visually, we depict P-patterns with green background, and N-patterns with red background. Figure 5(b) shows an N-pattern stating that non-persistent classes should not be related to tables. In this way, PAMOMO specifications can indicate not only how elements should be related, but also forbidden situations, i.e., it supports a non-constructive specification style. Finally, the P-pattern in Figure 5(c) has a disabling condition with label `N(Parent)`. The main CT-graph, with label `ClassTable`, demands a trace relating persistent classes and tables with same name. The disabling condition forbids the existence of a parent for the class. Altogether, the trace between the class and the table is only required to occur when the class does not have a parent.

An inter-modelling specification is made of a set of declarative patterns. As we will explain in the following sub-section, patterns are given different semantics depending on the usage scenario (forward/backward transformation, matching, traceability, etc.). The idea is to interpret patterns as special graph constraints [EEPT06] that should hold (for P-pattern) or not (for N-patterns) in the CT-graph where we evaluate the specification.

3.3 Satisfaction checking

Patterns are interpreted differently depending on the inter-modelling scenario [GLO11]. For instance, if our purpose is performing forward batch transformation, then the P-pattern in Figure 5(a) is interpreted as *"for each persistent package, there must be a schema with same name"*. Conversely, in backward transformation, the pattern is interpreted as *"for each schema, there*



(a) Forward satisfaction ($\texttt{SAT}_F$) (b) Match satisfaction ($\texttt{SAT}_M$)

Figure 6: Different notions of satisfaction.

must be a persistent package with same name". The model matching and model traceability scenarios consider the source and target of patterns at the same time. For instance, in model matching, the same pattern is interpreted as *"for each package and schema with same name, there should be a trace relating them"*. Note that, in all cases, there is a part of the pattern that is sought with a "for-all" iteration, and for each occurrence, the existence of the whole pattern is demanded. However, depending on the specific inter-modelling scenario, the domain of the sought part (source, target or both) and how this part is constructed differs.

In forward satisfaction (i.e. in the forward transformation scenario), the sought part is called *forward pre-condition*. This is built by glueing the enabling condition of the pattern (if there is any), together with the source part of the main CT-graph. Moreover, we built so-called negative forward pre-conditions by restricting each disabling condition of the pattern to the source part (see [GLO09] for the technical details). This construction is illustrated in Figure 6(a). As pattern `ClassTable` does not have an enabling condition, the forward pre-condition is equal to the source CT-graph. The negative forward pre-condition is made of the source of the pattern's disabling condition.

A CT-graph M forward-satisfies a positive pattern P, written $M \models_{SAT_F} P$, if for each occurrence of P's forward pre-condition such that there is no occurrence of P's forward negative pre-conditions, there is some occurrence of P's main CT-graph. These conditions are expressed using CT-morphisms, as Figure 6(a) shows. In the figure, there is only one morphism from the forward pre-condition that does not commute with a morphism from the forward negative pre-condition. This morphism is the one identifying `c` in the forward pre-condition and the model. For this morphism, there is a commuting morphism from the pattern's main constraint. Thus, the model forward-satisfies the pattern.

The forward satisfaction of a negative pattern is similar. In this case, for each occurrence of its forward pre-condition, there should not be an occurrence of its main CT-graph. The definition of backward satisfaction for patterns is symmetric to the forward case.

A CT-graph M forward-satisfies a specification S, written $M \models_{SAT_F} S$, if it forward-satisfies all the patterns in S. A pattern P spans a language $SEM_F(P) = \{M \mid M \models_{SAT_F} P\}$, which is the set of all CT-graphs that satisfy the pattern. Therefore, the semantics of PAMOMO is composi-

tional, as the semantics of a specification S is the intersection of the languages of all its patterns, $SEM_F(S) = \bigcap_{P \in S} SEM_F(P)$.

In model matching, we use a different notion of satisfaction in which, first, the so-called *match pre-condition* is sought. The match pre-condition of a pattern is made of the source and target parts of its main CT-graph, glued together with the pattern's enabling condition (if there is any). Moreover, we built so-called *negative match pre-conditions* by restricting each disabling condition of the pattern to its source and target parts. This construction is illustrated in Figure 6(b). The satisfaction checking procedure is similar as before, as for each occurrence of the match pre-condition that does not commute with any occurrence of the match negative pre-conditions, there should be a commuting occurrence of the pattern's main CT-graph. In Figure 6(b), the CT-graph M match-satisfies the pattern, written $M \models_{SAT_M} ClassTable$, as for each combination of class and table equally named, a trace exists.

The notions of satisfaction presented so far make sure that the source, target and trace models contain all necessary elements according to a specification. However, they do not check whether unnecessary source and target elements or whether incorrect traces exist in the models. This closed-world assumption for a given specification can also be considered in order to detect incorrect traces that can be subsequently deleted by an operational mechanism, or else change the source or target models to achieve consistency.

In practice, we generate OCL code from the patterns and use it against a given CT-graph to check whether it satisfies a specification [GLKP10a]. As an example, Listing 1 shows the generated OCL code to check forward satisfaction of pattern `ClassTable`.

```
1  operation forwardsat_ClassTable() : Boolean {
2    return Class.allInstances().forAll(c | c.persistent=true and not Class.allInstances().exists(p | c.parent.includes(p) )
3           implies Table.allInstances().exists(t |
4                   C2T.allInstances().exists(m | m.source=c and m.target=t and checkatt_ClassTable(c, t, m)));
5  }
6  operation checkatt_ClassTable( c:Class, t:Table, m:C2T ) : Boolean {
7    var X:=c.name;
8    var Y:=t.name;
9    return Y=X;
10 }
```

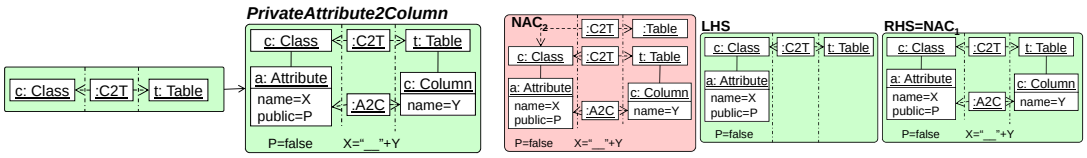
Listing 1: OCL code to check forward satisfaction of pattern in Figure 5(c).

Once we have seen different notions of satisfaction, the next subsection overviews the generation of operational mechanisms to enforce the patterns.

3.4 Generation of operational mechanisms

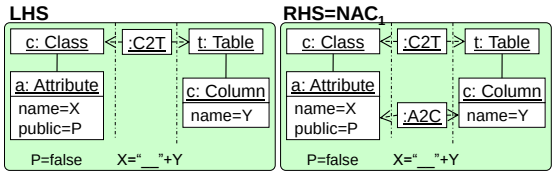
Each inter-modelling scenario requires a different operational mechanism. For example, in forward transformation, the target and trace models are created from scratch starting from the source model. In model matching and model traceability, a trace model is built mapping elements in the source and target models.

We implement these operational mechanisms by generating operational TGG rules from the patterns in the specification [GLO09], using a similar procedure to the generation of operational TGG rules from declarative TGG rules [Sch94]. In our case, a TGG operational rule is a non-deleting rule, made of a left-hand side (LHS) CT-graph L, a right-hand side (RHS) CT-graph R, a CT-morphism $r\colon L \to R$, a set $NAC_{pre} = \{n_i\colon L \to N_i\}$ of negative application conditions

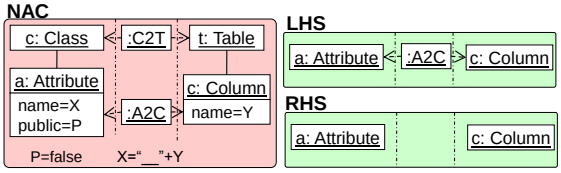


(a) A pattern with enabling condition

(b) Generated forward rule



(c) Generated model matching rule



(d) Generated trace deleting rule

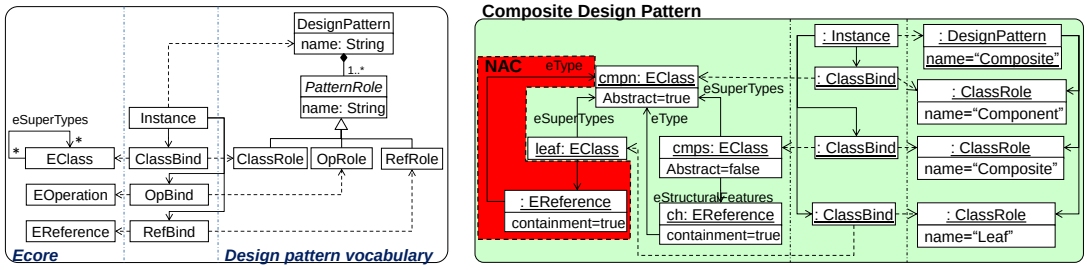
Figure 7: Generation of operational TGG rules.

and a set $NAC_{post} = \{n_j : R \to N_j\}$ of negative post-conditions. A rule can be applied to a CT-graph M if a CT-morphism $m : L \to M$ exists, and there is no commuting morphism from any negative application condition in NAC_{pre}. If such a CT-morphism exists, the rule can be applied by glueing M and R through its common elements in L (i.e. by a pushout as seen in Figure 4). As a result, M is enlarged with the new elements in R, and the formula in R is added to M as well. After applying the rule, the negative post-conditions in NAC_{post} are checked, and if some of them are found in the resulting CT-graph, then the rule application is undone[1]. A grammar is made of a set of rules. Applying a grammar means an iterated application of its rules (chosen in random order) until no rule is further applicable.

In our case, if the inter-modelling scenario is forward transformation, then we generate a rule from each P-pattern. The LHS of the rule contains the forward pre-condition of the pattern (i.e. the source of its main CT-graph together with the enabling condition). The RHS of the rule contains the main CT-graph. A number of negative application conditions are added to the rule to ensure termination, as explained in [GLO09]. Moreover, if the specification contains N-patterns, then these are transformed into negative post-conditions of the generated rule. As an example, Figure 7(a) shows a P-pattern with an enabling condition, and Figure 7(b) contains the forward transformation rule generated from the pattern.

Our generation procedure yields grammars that are terminating (i.e. their execution eventually stops) and correct (i.e. each generated terminal CT-graph forward-satisfies the specification) [OGLE09]. However, in order to achieve completeness (i.e. being able to generate all possible models that satisfy the specification) we need to generate additional rules with increasingly

[1] In practice, this is usually checked a priori by *advancing* the post-conditions to pre-conditions.



(a) Meta-model for design pattern discovery (b) Specifying the *Composite* design pattern

Figure 8: Inter-modelling for the discovery of design patterns.

bigger LHS. The generated grammar is in general not confluent because, indeed, a specification may admit several target models that are considered correct transformations of the same source model [OGLE09].

The generation of operational TGG rules to implement other inter-modelling scenarios like model matching and traceability is similar [GLO11]. As an example, Figure 7(c) shows the rule generated from the pattern in Figure 7(a) for model matching. Moreover, if we consider a closed-world assumption, then we need additional rules to delete incorrect traces between models. These rules can also be generated from the inter-modelling specifications. For instance, the rule in Figure 7(d) deletes an incorrect trace between an attribute and a column if the trace does not conform to any P-pattern in the specification. The NAC forbids deleting the trace in case the trace context is the one specified by the `PrivateAttribute2Column` P-pattern, assuming this is the only P-pattern containing such trace type.

4 Examples

Next, we present two applications of our inter-modelling approach.

4.1 Discovery of design patterns

We have implemented a general inter-modelling specification tool, based on PAMOMO, over the Eclipse Modeling Framework (EMF) [SBPM08]. In this tool, patterns can be specified using a textual syntax, and then can be compiled into OCL expressions and EOL programs that implement the notions of satisfaction and the behaviour of the operational TGG rules presented before. In this subsection we show an application of this tool to the discovery of design patterns on meta-models.

In the model matching scenario, an inter-model specifies similarity criteria between two possibly heterogeneous models. We have used this semantics to find and annotate occurrences of design patterns [GHJV94] in meta-models. In particular, the two models to compare are: (a) the meta-model and (b) a model with the design pattern roles, for each design pattern. In this case, each inter-modelling pattern in the specification corresponds to the description of a design pattern. Thus, whenever an occurrence of an inter-modelling pattern is found, the corresponding traces are created [GLKP10a] to indicate a design pattern instance in the meta-model.

Figure 8(a) shows an excerpt of the meta-models used for this example. The left part corresponds to a small fragment of the ecore meta-model, while the right part contains the vocabulary meta-model used to specify the roles of the elements (classes, operations, references) involved in design patterns. The trace meta-model in between permits assigning design pattern roles to the elements in the ecore meta-model. For this example, we used an extended theory where traces may point to just one element in one of the models (e.g., node `Instance`) and edges in the trace model are not mapped [GLKP10a]. Then, each design pattern is specified as a PAMOMO inter-modelling pattern. Figure 8(b) shows a simplified version of the *Composite* design pattern [GHJV94]. There is a disabling condition, which is depicted using a compact notation enclosed in a polygon labelled as NAC together with the main CT-graph. The disabling condition forbids containment references from the `Leaf` class to the `Component` class.

To identify design patterns in a meta-model, we have to give as input the ecore meta-model of interest, as well as an instance of the meta-model to the right of Figure 8(a). The matching mechanism generates traces identifying the occurrences of the different design patterns specified in the inter-modelling patterns. Such traces can be maintained correct by the generated operational mechanisms, creating new traces whenever a new instance of a design pattern is created, and deleting incorrect traces whenever some meta-model change deletes a design pattern instance.

4.2 Transformation contracts for automated testing

The most widely adopted means to build transformations is by the use of transformation implementation languages like ATL [JABK08], ETL [KPP08] or simply Java. These approaches are likely the most immediate to build transformations, as they are well supported by development environments. However, their abstraction level is close to programming languages, so that transformations often become difficult to program, test, maintain and understand.

To simplify the previous tasks, in previous works we have proposed using inter-models as a way to specify *requirements* for model transformations. In particular, we use PAMOMO specifications as contracts for model transformation implementations, so that they can be used to automate their testing [GLW+12]. A PAMOMO specification can be used to specify: (a) preconditions, (b) post-conditions and (c) invariants that a transformation implementation needs to fulfil. Pre-conditions specify conditions that any input models to the transformation must satisfy. They are specified as PAMOMO patterns where the correspondence and target parts of the main CT-graph is empty. Post-conditions specify conditions that every output model resulting from the transformation must satisfy. They are specified as PAMOMO patterns where the source and the correspondence parts of the main CT-graph is empty. Invariants specify properties that pairs of source/target models should satisfy (i.e. patterns where the source and target are not empty).

Figure 9 shows a scheme of our approach. First, the transformation designer can gather requirements for the transformation in the form of PAMOMO patterns. These patterns specify *what* the transformation is to do. Then, an *implementation* could be constructed using any transformation language (e.g. ATL, ETL or simply Java)[2]. This implementation can be tested against its requirements. For this purpose, we have currently two approaches. In the first one, we generate

[2] Although we can produce operational mechanisms (operational TGGs) for model transformation from our specifications, we do not currently have a full implementation, which would rely on a combination of rewriting and constraint solving.

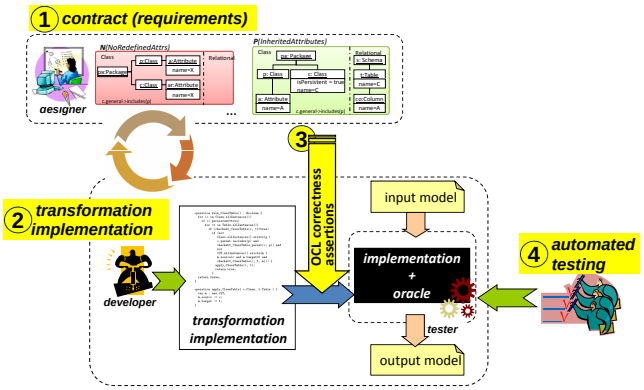


Figure 9: Automated testing with PAMOMO

OCL expressions from the specification, which are evaluated before and after the transformation execution [GLKP10b]. In particular, the pre-conditions are checked before the execution, whereas the post-conditions and invariants are checked after the execution. If the input/output models satisfy all generated OCL expressions (i.e. the expressions return *true*) then it means that the models satisfy the specification. In this way, the generated OCL code acts as a partial oracle function for transformation testing. In our second approach, we generate QVT-Relations code from the requirements (instead of OCL) [GLW+12]. The advantage is that executing the QVT-Relations code in check-only mode using an engine like ModelMorf provides more information about the reasons for failure than an OCL expression. In particular, the engine reports the location in the models where the specification is not fulfilled. Please note that, normally, PAMOMO contracts are under specifications of the behavior that transformations implementation should satisfy. Moreover, the meta-models used by the implementations can be refinements of those used in the specification.

As an example, Figure 10 shows a pattern capturing a transformation requirement: inherited attributes should be transformed into columns of the table created for the child class. The figure shows to the left the OCL code generated from the pattern, which checks the satisfaction of the pattern by a pair of models. To the right, the figure shows the QVT-Relations code generated from the same pattern for the same purpose. In both cases, it can be observed that the generated code is less compact than the original pattern [GLW+12].

5 Related Work

We define *inter-modelling* as the activity of modelling relations between models. Thus, we describe such relations as models, as opposed to hard-coding specific mechanisms for each concrete scenario. In general, models tend to be more flexible, understandable and maintainable than lower-level programs. Inter-models can be seen as a generalization of the term *transformation-model* [BBG+06], comprising additional inter-modelling scenarios other than transformations.

Our approach is closely related to Triple Graph Grammars (TGGs) [Sch94]. In TGGs, an inter-model is specified by a declarative TGG, from which operational TGG rules are derived for different scenarios, like forward/backward transformation or model matching. Two models are

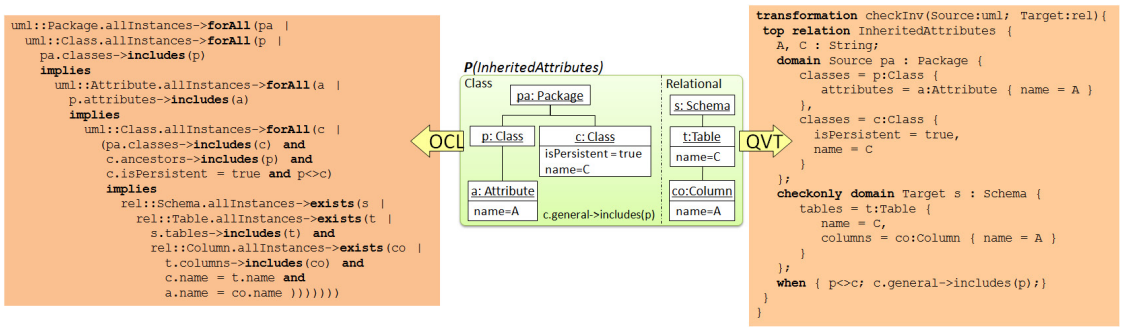


Figure 10: A pattern capturing a transformation requirement (middle), and its compilation into OCL (left) and QVT-Relations (right)

correctly related if they can be produced by the declarative TGG. Therefore, one must resort to parsing. In our case, an inter-model is made of constraints, and two models are correctly related *according to some scenario* if they satisfy all constraints in the specification. To our knowledge, there is no equivalent to our negative patterns in TGGs. In practice, we check whether a set of models satisfy a specification by deriving appropriate OCL expressions. Therefore, an advantage of our approach is that we have different notions of satisfaction depending on the scenario. Moreover, the generated OCL can be used in existing MDE tools. For instance, we can use existing OCL-based constraint solvers to generate models satisfying a specification, which may be useful for transformation testing [Gue12].

Another inter-modelling tool is the ATLAS Model Weaver (AMW) [FBJ+05], which is used to establish relationships (i.e. links) between models. The links are stored in a so-called weaving model, conforming to a weaving meta-model. Simple source-to-target transformations can be derived from a weaving model, but only when the source and target meta-models are very similar. The definition of complex conditions enabling the creation of traces, like those that can be encoded with PAMOMO, requires defining additional conditions at the model level by means of patterns of source and target instances, which is not supported in AMW.

In [DMC12], a formal framework for inter-modelling is proposed based on Kleisly categories. Instead of using patterns to specify relations between models, model queries are defined, which enrich the models with derived information. Then, trace models are direct mappings between the enlarged models. It is up to future work to identify how our patterns correspond to such queries, and how such framework can be used for the inter-modelling scenarios we have presented.

6 Conclusions and Future Work

In this paper, we have argued that many MDE activities that involve several models can be seen as a form of inter-modelling. We define inter-modelling as the activity of building models that describe how other models should be related. Inter-modelling specifications can be used in check-only mode to assert whether several models are correctly related with respect to the specifications, for different scenarios (e.g. model transformation, model matching and model traceability). They can be used operationally as well, so that the models can be manipulated to

enforce their conformance to the specifications for a certain scenario. We have presented the inter-modelling language PAMOMO, which follows a declarative, bi-directional, relational style. Finally, we have shown several examples illustrating its use in practice.

We are currently working on using PAMOMO to specify transformation requirements, and for transformation testing. In particular, we are exploring the automatic generation of input models for testing, covering all relevant properties of the transformation according to the specification. We are also working on providing tool support for using PAMOMO as a part of a family of languages, called *transML* [GLK+12], for the engineering of model transformations.

Acknowledgements: Work sponsored by the Spanish Ministry, with project "Go Lite" (TIN2011-24139), and the R&D programme of Madrid Region with project "e-Madrid" (S2009/TIC-1650).

Bibliography

[BBG+06] J. Bézivin, F. Büttner, M. Gogolla, F. Jouault, I. Kurtev, A. Lindow. Model transformations? Transformation models! In *MODELS'06*. LNCS 4199, pp. 440–453. Springer, 2006.

[DMC12] Z. Diskin, T. Maibaum, K. Czarnecki. Intermodeling, queries, and Kleisli categories. In *FASE'12*. LNCS 7212, pp. 163–177. Springer, 2012.

[EEPT06] H. Ehrig, K. Ehrig, U. Prange, G. Taentzer. *Fundamentals of algebraic graph transformation*. Springer-Verlag, 2006.

[FBJ+05] M. D. D. Fabro, J. Bézivin, F. Jouault, E. Breton, G. Gueltas. AMW: A generic model weaver. In *1éres Journées sur l'Ingénierie Dirigée par les Modéles*. 2005. See also http://www.eclipse.org/gmt/amw/.

[GHJV94] E. Gamma, R. Helm, R. Johnson, J. M. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.

[GL12] E. Guerra, J. de Lara. An algebraic semantics for QVT-Relations check-only transformations. *Fundam. Inform.* 114(1):73–101, 2012.

[GLK+12] E. Guerra, J. de Lara, D. S. Kolovos, R. F. Paige, O. M. dos Santos. Engineering model transformations with transML. *Software and Systems Modeling* In press, 2012.

[GLKP10a] E. Guerra, J. de Lara, D. S. Kolovos, R. F. Paige. Inter-modelling: From theory to practice. In *MoDELS (1)*. LNCS 6394, pp. 376–391. Springer, 2010.

[GLKP10b] E. Guerra, J. de Lara, D. S. Kolovos, R. F. Paige. A visual specification language for model-to-model transformations. In *VLHCC'10*. Pp. 119–126. IEEE CS, 2010.

[GLO09] E. Guerra, J. de Lara, F. Orejas. Pattern-based model-to-model transformation: Handling attribute conditions. In *ICMT'09*. LNCS 5563, pp. 83–99. Springer, 2009.

[GLO11] E. Guerra, J. de Lara, F. Orejas. Inter-modelling with patterns. *Software and Systems Modeling* In press, 2011.

[GLW+12] E. Guerra, J. de Lara, M. Wimmer, G. Kappel, A. Kusel, W. Retschitzegger, J. Schönböck, W. Schwinger. Automated verification of model transformations based on visual contracts. *Automated Software Engineering Journal* In press, 2012.

[Gue12] E. Guerra. Specification-driven test generation for model transformations. In *ICMT'12*. LNCS 7307, pp. 40–55. Springer, 2012.

[JABK08] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev. ATL: A model transformation tool. *Science of Computer Programming* 72(1-2):31 – 39, 2008. See also http://www.emn.fr/z-info/atlanmod/index.php/Main_Page. Last accessed: Nov. 2010.

[Kol09] D. S. Kolovos. Establishing correspondences between models with the Epsilon Comparison Language. In *ECMDA-FA'09*. LNCS 5562, pp. 146–157. Springer, 2009.

[KPP06] D. S. Kolovos, R. F. Paige, F. Polack. The Epsilon Object Language (EOL). In *ECMDA-FA'06*. LNCS 4066, pp. 128–142. Springer, 2006.

[KPP08] D. S. Kolovos, R. F. Paige, F. Polack. The Epsilon Transformation Language. In *ICMT'08*. LNCS 5063, pp. 46–60. Springer, 2008.

[LG08] J. de Lara, E. Guerra. Pattern-based model-to-model transformation. In *ICGT*. LNCS 5214, pp. 426–441. 2008.

[Mod] ModelMorf. http://www.tcs-trddc.com/trddc_website/scripts/project_detail.php?lab=SWRD&project_id=44. Last accessed: April 2012.

[OCL] OCL. http://www.omg.org/spec/OCL/2.3.1/.

[OGLE09] F. Orejas, E. Guerra, J. de Lara, H. Ehrig. Correctness, Completeness and Termination of Pattern-Based Model-to-Model Transformation. In *CALCO'09*. LNCS 5728, pp. 383–397. Springer, 2009.

[PDK+11] R. F. Paige, N. Drivalos, D. S. Kolovos, K. J. Fernandes, C. Power, G. K. Olsen, S. Zschaler. Rigorous identification and encoding of trace-links in model-driven engineering. *Software and System Modeling* 10(4):469–487, 2011.

[QVT] QVT. http://www.omg.org/docs/ptc/05-11-01.pdf.

[SBPM08] D. Steinberg, F. Budinsky, M. Paternostro, E. Merks. *EMF: Eclipse Modeling Framework, 2nd Edition*. Addison-Wesley Professional, 2008.

[Sch94] A. Schürr. Specification of graph translators with triple graph grammars. In *WG'94*. LNCS 903, pp. 151–163. Springer, 1994.

[WP10] S. Winkler, J. von Pilgrim. A survey of traceability in requirements engineering and model-driven development. *Software and System Modeling* 9(4):529–565, 2010.