Noname manuscript No.
(will be inserted by the editor)

Automated Verification of Model Transformations based on Visual Contracts

Esther Guerra · Juan de Lara · Manuel Wimmer · Gerti Kappel · Angelika Kusel · Werner Retschitzegger · Johannes Schönböck · Wieland Schwinger

Received: date / Accepted: date

Abstract Model-Driven Engineering promotes the use of models to conduct the different phases of the software development. In this way, models are transformed between different languages and notations until code is generated for the final application. Hence, the construction of *correct* Model-to-Model (M2M) transformations becomes a crucial aspect in this approach.

Even though many languages and tools have been proposed to build and execute M2M transformations, there is scarce support to specify correctness requirements for such transformations in an implementation-independent way, i.e., irrespective of the actual transformation language used.

In this paper we fill this gap by proposing a declarative language for the specification of visual contracts, enabling the verification of transformations defined with any transformation language. The verification is performed by compiling the contracts into QVT to detect disconformities of transformation results with respect to the contracts. As a proof of concept, we also report on a graphical modeling environment for the specification of contracts, and on its use for the verification of transformations in several case studies.

Keywords Model-Driven Engineering · Model Transformation · Contract-based Specification · Verification · QVT-Relations

This work has been funded by the Austrian Science Fund (FWF) under grant P21374-N13, the Spanish Ministry of Science under grants TIN2008-02081 and TIN2011-24139, and the R&D programme of the Madrid Region under project S2009/TIC-1650.

Esther Guerra · Juan de Lara
Universidad Autónoma de Madrid, Spain
E-mail: firstname.lastname@uam.es

Manuel Wimmer · Gerti Kappel · Johannes Schönböck
Vienna University of Technology, Austria
E-mail: lastname@big.tuwien.ac.at

Angelika Kusel · Werner Retschitzegger · Wieland Schwinger
Johannes Kepler University Linz, Austria
E-mail: firstname.lastname@jku.at

1 Introduction

Model-Driven Engineering (MDE) [46] proposes an active use of models to conduct the different phases of software development. Hence, models become first-class artifacts throughout the software lifecycle, which leads to a shift from the "everything is an object" paradigm to the "everything is a model" paradigm [6]. In this context, model transformations are crucial for the success of MDE, being comparable in role and importance to compilers for high-level programming languages, since models have to be automatically refined until the code of the final application is obtained. Thus, in MDE there is a recurring need to transform models between different languages and abstraction levels, e.g., to migrate between language versions, to translate models into semantic domains for analysis, to generate platform-dependent from platform-independent models, or to refine and abstract models [18]. These kinds of transformations are called Model-to-Model (M2M) transformations, and one of the major challenges of MDE is their automation while ensuring the correctness of the produced models.

M2M transformations are usually defined with dedicated languages tailored for the task of transforming models, like QVT [43], ATL [27] or ETL [29] (cf. [13] for a detailed overview). Most of these languages have a strong focus on the implementation of transformations but miss to provide means for their analysis, design and verification. However, just like any other software, transformations should be engineered using sound, robust engineering techniques [18, 23]. This necessity is even more acute given the prominent role of transformations in MDE, and their use in increasingly complex scenarios. Hence, the MDE community demands for methods, notations and techniques supporting appropriate *abstractions* to be used in the different phases of the transformation development and, in particular, for the verification of transformations.

In order to fill this gap, in [22, 23] we introduced a visual, declarative, formal *specification* language to describe, in an implementation-independent way, correctness requirements of the transformations and of their input and output models. This language, called PaMoMo (Pattern-based Modeling Language for Model Transformations), was initially designed to play a similar role for M2M transformations to the role that Z [48] or Alloy [24] play for general software development: specifying properties that a transformation should fulfill, regardless of its particular implementation. Thus, PaMoMo specifications express *what* a transformation should do, but not *how* it should be done, providing an adequate level of abstraction to express transformation requirements. These requirements may correspond to preconditions, postconditions or invariants that the input and output models of a transformation as well as the transformation itself should fulfill. Even though some researchers [20, 34, 40] have proposed the use of OCL for this task, in this paper we show the benefits of using of a visual, formal, bidirectional and domain-specific notation like PaMoMo. In particular, we argue that PaMoMo patterns lead to more succinct specifications, enable reasoning at the pattern level, and permit more informative feedback (higher diagnosability).

In this paper, we extend the expressivity of PaMoMo with support to define sets of variable size in invariants as well as enabling and disabling conditions in pre- and postconditions. We also present a set of reasoning rules aimed at detecting redundancies, contradictions and potential errors in specifications. More important, borrowing ideas from the *design by contract* approach [39], we use PaMoMo to specify contracts for the automated verification of transformation implementations by the compilation of these contracts into executable transformations expressed in the QVT-Relations language [43]. These transformations are executed before the transformation under test (to check the preconditions) and afterwards (to check invariants and postconditions), and provide the user with detailed information on which contracts were violated (if any) and where. In this way, the present work improves the feedback returned to users with respect to a previous compilation of our patterns into OCL presented in [22], as in OCL we were only able to report whether a pattern was satisfied or not, whereas now we report in addition the parts of the models that make a contract fail. Finally, we also present the new PaCo-Checker tool (PaMoMo Contract-Checker) which supports the visual specification of contracts, their compilation into QVT-Relations, its chaining with the execution of the transformation under test, and the visualization of the test results. We illustrate the usefulness of our method on a number of case studies.

The rest of the paper is organized as follows. Section 2 introduces M2M transformation and presents a running example that we will use throughout the paper. Section 3 introduces the main concepts of contract-based specification as well as our approach to model transformation contracts. Section 4 presents our specification language PaMoMo. Section 5 recalls the main concepts of QVT-Relations as this is the target language for the compilation of PaMoMo, whereas Section 6 details this compilation. Section 7 describes how to conduct automated verification with the PaCo-Checker tool. Section 8 illustrates further features of our approach on a number of case studies. Finally, Section 9 compares with related work and Section 10 concludes the paper.

2 Background: Model Transformation in a Nutshell

In this section we present the main concepts of M2M transformations, and then introduce an example that will be used throughout the paper.

2.1 Model-to-Model Transformations

Models play a fundamental role in MDE, and hence their transformation is crucial for the success of MDE. Models in this context are abstractions of systems and/or their environments [13]. In the same way as programs have to follow certain syntactic constraints – commonly described by grammars –

models also have to follow syntactic constraints given by so-called metamodels which define their abstract syntax [32]. Thus, in order to describe how models should be transformed into other models, the transformation definition uses the respective metamodels the models conform to (cf. Fig. 1). Such definitions are finally executed by dedicated transformation engines.

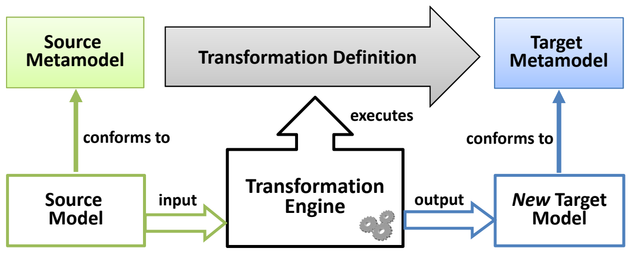


Fig. 1 Model transformation: exogenous, batch scenario

Fig. 1 shows a *batch and exogenous* transformation scenario [38], where a source model conformant to a source metamodel, is transformed into a target model conformant to a target metamodel. Many other scenarios are possible as well. First, the source model may change after the transformation is executed. In this case, it is sometimes more efficient not to build the target model from scratch but to *update* it. Then, transformations can also be *bidirectional*, if the same specification can be used to transform from source to target and the other way round. Transformations can also be used in *check-only* mode, to ascertain whether two existing models comply with the transformation definition. Finally, a model may be transformed *in-place*, for example for refactoring. In this case, the transformation is called *endogenous* and it only considers one metamodel. In this paper, we target *batch and exogenous* transformations, but our specification language can be used to specify correctness requirements for other transformation scenarios as well.

2.2 Transformation Scenario: From Class Diagrams to Relational Schemas

Before delving into details, we introduce a concrete transformation scenario that is used throughout the paper. In particular, we present a small extract of the well-known `Class2Relational` transformation (cf. Fig. 2) [7], which has been chosen due to its popularity. The metamodel to the left of the figure is used to represent a simple object oriented modeling language. While the `parents` reference contains direct ancestors of classes, the derived `ancestors` reference contains the transitive closure of `parents`, and therefore it includes indirect ancestors as well. The metamodel to the right is used to represent a language for defining database schemas. In this scenario, the goal is to transform instances of the `class` metamodel into instances of the `relational` metamodel. For this transformation, six main requirements arise:

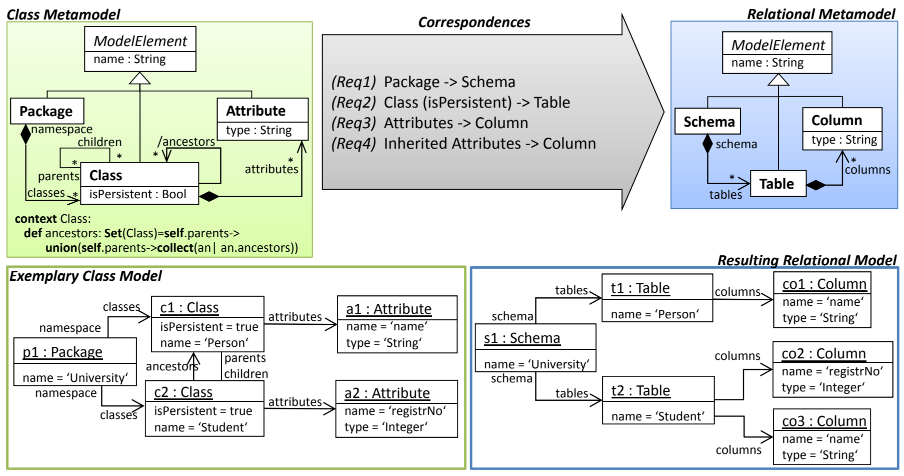


Fig. 2 Running example

– *Requirement 1*: For each instance of the metaclass `Package`, a corresponding instance of the metaclass `Schema` should be generated, which should be equally named (cf. instances `p1` and `s1` in Fig. 2).
– *Requirement 2*: For each persistent instance of the metaclass `Class`, an instance of the metaclass `Table` should be generated, which should be equally named (cf. instances `c1`, `c2` and `t1`, `t2` in Fig. 2). The table should be added to the schema created from the package the class belongs to.
– *Requirement 3*: For each instance of the metaclass `Attribute` that belongs to a persistent class, an equally named instance of the metaclass `Column` should be generated, and this should be added to the table created from the owner class (cf. instances `a1`, `a2` and `co1`, `co2` in Fig. 2).
– *Requirement 4*: Since the relational metamodel does not support inheritance between tables and since information loss should be prevented during the transformation process, for each inherited attribute a corresponding `Column` instance should be generated (cf. instance `co3` in Fig. 2).

Besides requirements that any pair of input/output models should satisfy, some requirements may solely apply to the input models. Such requirements are used to add *further constraints on the input models* in order to exclude those not handled by the transformation (although they conform to the source metamodel). This is due to the fact that metamodels allow for many different valid models, but a certain transformation might only cover a subset thereof. A requirement on the input models of our example is the following:

– *Requirement 5*: Class models cannot contain redefined attributes (i.e., attributes with the same name in an inheritance hierarchy), since otherwise tables containing equally named columns would result.

Finally, a certain transformation might need to guarantee that the produced output models fulfill certain conditions (beyond metamodel constraints). In our example, we demand the following:

– *Requirement 6*: Relational models cannot contain tables with equally named columns, even though this is allowed by the metamodel.

3 Model Transformation Contracts

In order to make the previous requirements explicit, we propose their specification by contracts. Therefore, we next discuss different usages of contracts for M2M transformations, and then introduce PaMoMo for their specification.

3.1 Increasing Quality through Design by Contract

Design by contract [39] was introduced as a means to increase quality in terms of correctness and robustness of the constructed software. One of the advantages of contracts is that they allow defining *what* a piece of software does but not *how* it is done. Different levels of contracts can be distinguished comprising *syntactic contracts* and *behavioral semantic contracts* [5]. The former enforce syntactically valid programs. In the context of model transformations, syntactic contracts are specified by the source and target metamodels since they describe the types of the manipulated data, implying that the source and target models must conform to these types [40]. In contrast, behavioral semantic contracts put further restrictions on the required input models, the produced output models as well as their combinations [40]. In this way, in the first place, behavioral semantic contracts can be used to *precisely* specify the conditions (going beyond metamodel constraints) to be satisfied by input models such that the transformation is applicable, i.e., *preconditions*. Second, they can be used to express that an output model should or should not contain certain configurations of elements, i.e., *postconditions*. Finally, they can be used to specify *what* conditions need to be satisfied by any pair of input/output models of a correct transformation, i.e., *invariants* of the transformation.

In the context of model transformations, contracts can be useful in several scenarios [12]:

– **Implementation:** A contract is a useful document for the transformation designer in the development phase, to make explicit the requirements that need to be implemented in a transformation.
– **Documentation:** Contracts serve as a useful documentation of the transformation in the maintenance phase. Moreover, if contracts have a formal semantics, they can be used to select transformations by matching properties of a required transformation and properties of transformations stored in a transformation library.
– **Compatibility Checking:** Contracts can be used to check the compatibility of transformations in a chaining scenario, e.g., to check whether the postconditions of a preceding transformation are compatible with the preconditions of a succeeding transformation.

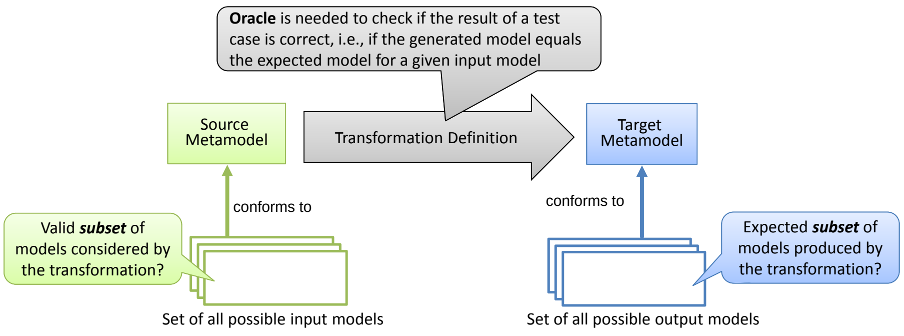

Oracle is needed to check if the result of a test case is correct, i.e., if the generated model equals the expected model for a given input model
Source Metamodel
Transformation Definition
Target Metamodel
conforms to
conforms to
Valid ***subset*** of models considered by the transformation?
Expected ***subset*** of models produced by the transformation?
Set of all possible input models
Set of all possible output models

Fig. 3 Model transformation testing challenge

– **Testing:** A common need in model transformation testing is to automatically compare expected output models to generated output models [37]. Unfortunately, the oracle that should predict the expected output models remains a major challenge [3], for which contracts (invariants) could be used to partially determine the expected output model.

The contracts specified using PaMoMo can be beneficial in each of the above discussed scenarios. Nevertheless, the focus in this paper is on the testing scenario, i.e., how preconditions, invariants and postconditions can be applied to test model transformations (cf. Fig 3).

3.2 Model Transformation Contracts with PaMoMo at a Glance

Contracts may be realized by being embedded in a certain language (e.g., assertions in Java) or described by a dedicated external language (e.g., Z [48] or Alloy [24]). The realization by a dedicated language has two main advantages though: (i) the definition of contracts is not tied to a particular target transformation language, i.e., it is *implementation-independent* (which is especially favorable in MDE since no dedicated standard transformation language has been brought forward so far [13]) and (ii) designers of transformations can make explicit desired properties of a transformation *before* implementation which would allow for test-driven development of transformations.

Thus, we adopt a declarative, formal, visual language called PaMoMo [22, 23] to express behavioral semantic contracts for M2M transformations in an implementation-independent way [22]. Fig. 4 outlines our approach. First, the transformation designer uses PaMoMo to define a contract specifying preconditions, postconditions, and invariants for the transformation (label 1). This contract has a formal semantics and can be analysed to discover redundancies, contradictions, and to measure coverage of the involved metamodels. Next, the developer can make use of the contract as a high-level model to implement the transformation (label 2). This implementation can be tested by compiling the contract into the executable QVT-Relations language (label 3), and then using a QVT engine in check-only mode in order to check the consistency of the

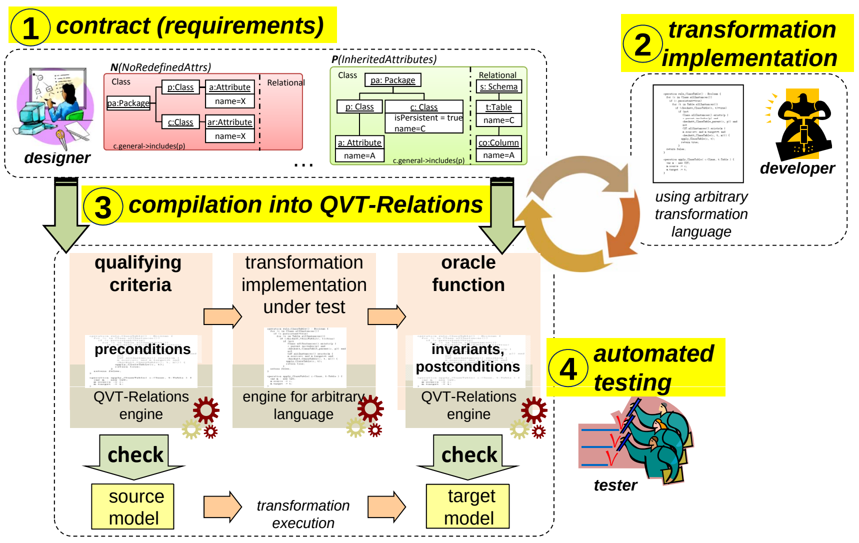


Fig. 4 Automated verification of transformations using PaMoMo

transformed models with respect to the contract. In this mode, a transformation is not used to produce a target model, but to check if a set of existing models conform to the transformation, and to report the locations where this is not the case by means of a built-in tracing mechanism. Hence, the compiled contract acts as an oracle describing invariants that output models should satisfy, and is used for automated testing (label 4). The compilation is also used to test whether a model can be used as input for the transformation.

Altogether, our approach to testing proceeds by chaining a QVT-Relations check-only transformation, derived from the preconditions in the contract, which checks the validity of the input model; next executing the transformation implementation; and finally checking that the input and resulting output models conform to the contract by using another check-only transformation.

In the following section we focus on the first step in our approach, namely the specification of contracts with PaMoMo.

4 Contract Specification with PaMoMo

In this section, we provide an overview of the syntax and semantics of PaMoMo (we refer to [22] for details on its formal semantics). We first describe how to specify contracts with PaMoMo, and then continue describing pattern reasoning rules to discover redundancies and conflicts in contracts.

4.1 Modeling Contracts with PaMoMo

A PaMoMo contract consists of a set of declarative visual patterns, which can be either *positive* or *negative*. Positive patterns describe necessary con-

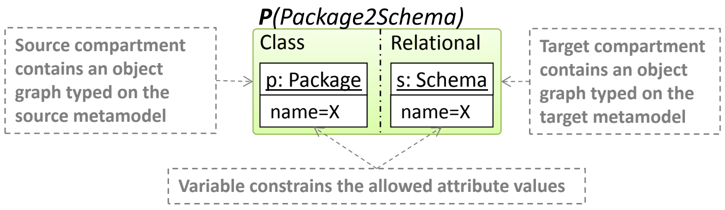

P(Package2Schema)
Class
Relational
p: Package
name=X
s: Schema
name=X
Source compartment contains an object graph typed on the source metamodel
Target compartment contains an object graph typed on the target metamodel
Variable constrains the allowed attribute values

Fig. 5 Positive invariant formalizing *requirement 1*

ditions to happen (i.e., the pattern is satisfied by a pair of models if these contain certain elements) while negative ones state forbidden situations (i.e., the pattern is satisfied if certain elements are not found). Patterns are bidirectional and can be interpreted forwards (e.g., to verify a source-to-target transformation) and backwards (e.g., to verify a target-to-source transformation). By default, we assume a forward semantics. Patterns are made of two compartments containing object graphs, plus a constraint expression using the Object Constraint Language (OCL) [42]. The left compartment contains objects typed on the source metamodel (e.g., `class`), while the objects to the right are typed on the target metamodel (e.g., `relational`).

As an example, Fig. 5 shows a positive pattern formalizing requirement 1 of the example transformation. We depict positive patterns in green with its name enclosed in `P(...)`, while negative patterns are shown in red with its name enclosed in `N(...)`. Patterns where both the source and target compartments are not empty are called *invariants*.

Hence, patterns are made of a graphical part, specified visually, and a textual expression enabling the specification of additional constraints. Objects in the source and target compartments may have attributes that can be assigned either a concrete value or a variable (like X in the example). A variable can be assigned to several attributes to ensure equality of their values, or be used in the pattern constraint expression. This expression may involve elements of the source and target compartments. The invariant of Fig. 5 has no expression, but variable X is assigned to the name of the package and the schema, hence requiring the equality of both names.

Fig. 6 shows a scheme of the satisfaction of a positive and a negative invariant over a pair of models, where `EXP` represents the pattern constraint expression. Thus, the satisfaction for positive invariants amounts to check:

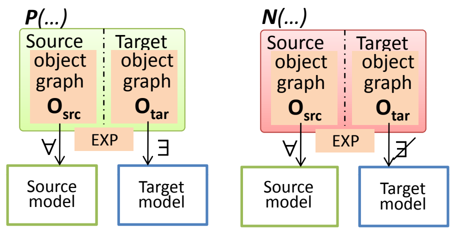

P(...)
Source object graph O_{src}
Target object graph O_{tar}
EXP
∀
∃
Source model
Target model
N(...)
Source object graph O_{src}
Target object graph O_{tar}
EXP
∀
∄
Source model
Target model

Fig. 6 Scheme of the semantics of positive and negative invariants

$$\forall Occ(O_{src}) \; s.t. \; EXP|_{src}(Occ(O_{src}))$$
$$\exists Occ(O_{tar}) \; s.t. \; EXP(Occ(O_{src}), Occ(O_{tar}))$$

where $EXP|_{src}$ is the part of the expression EXP that contains source objects, attributes and variables only, and $Occ(O_{src})$, $Occ(O_{tar})$ represent an occurrence of the source and target object graphs respectively. An occurrence is a binding from the objects in the object graph of the pattern to elements in the model. A pattern invariant is therefore satisfied either if we do not find an occurrence of the source object graph of the pattern (called *vacuous satisfaction*) or if for each occurrence of the source object graph, we find a corresponding occurrence of the target object graph (or do not find any if the invariant is negative). A contract is satisfied if *all* its patterns are satisfied, hence a *conjunction* is assumed between all the patterns of the contract.

Fig. 7 shows the invariants addressing requirements 2, 3 and 4 in our running example (i.e., transformation of classes, attributes and inherited attributes). The invariant to the left states that for each persistent class `c` in a package `p`, there must be an equally named table `t` in a corresponding schema `s`. The invariant in the middle states that each attribute `a` of a persistent class must be transformed into a column `co` with the same name and type. Finally, the right-most invariant states that if a class `c` has an ancestor class `p` owning an attribute `a`, then the table `t` that corresponds to `c` must contain a column with the same name as the attribute. This invariant contains a constraint expression checking that the derived property `ancestors` of class `c` includes the class `p` (i.e., `p` is a superclass of `c`).

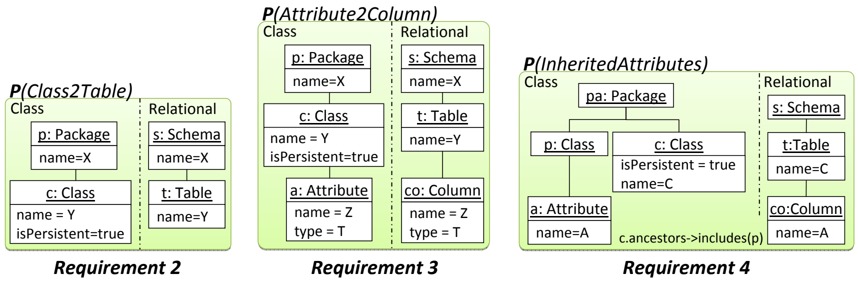


Fig. 7 Additional invariants formalizing *requirements 2*, *3* and *4*

4.1.1 Preconditions and Postconditions

In contrast to invariants, which relate source and target models, preconditions refer only to elements of the source metamodel (i.e., only the source compartment of the pattern contains an object graph) and postconditions refer only to elements of the target metamodel (i.e., only the target compartment

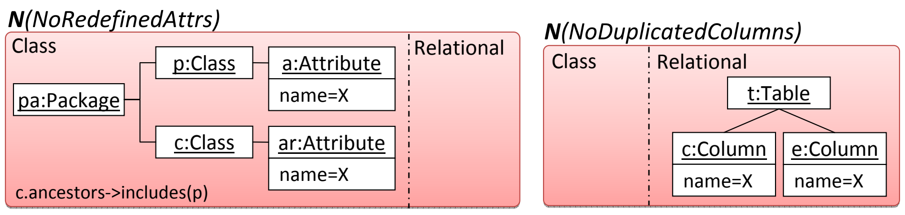


Fig. 8 Precondition (*requirement 5*) and postcondition (*requirement 6*)

contains an object graph). The left side of Fig. 8 shows a precondition expressing requirement 5 in our example (i.e., absence of redefined attributes in class hierarchies) by a negative pattern. The right part of the figure shows the postcondition to express requirement 6 (i.e., absence of duplicated columns in the same table) as a negated pattern as well.

Fig. 9 depicts a schema of the semantics of positive and negative preconditions. Positive preconditions demand the existence of a structure in the source model satisfying the expression constraint. Negative preconditions demand the absence of a structure in the source model satisfying the constraint expression. Postconditions have similar semantics, but are evaluated on the target model.

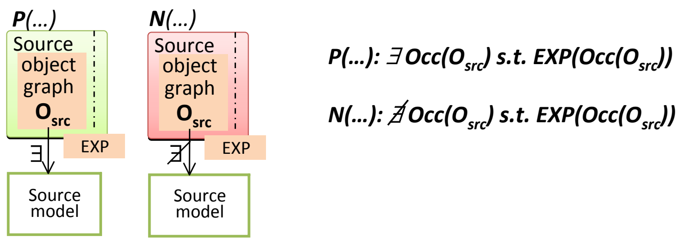


Fig. 9 Scheme of the semantics of positive and negative preconditions

4.1.2 Enabling and Disabling Conditions

The invariants we have presented so far check that for all occurrences of an object graph in the source model, a corresponding structure in the target exists. However, some more flexibility is often needed, to demand the satisfaction of a pattern only when certain conditions in the source *and* the target occur. For this purpose, patterns can define *enabling* and *disabling* conditions, which restrict their satisfaction context.

In particular, enabling and disabling conditions allow expressing properties with the form of an implication. Each pattern can define any number of disabling conditions and one enabling condition. This permits formulating properties of the form $if\ \langle enabling\rangle\ and\ (not\ \langle disabling_1\rangle)\ ...\ and\ (not\ \langle disabling_n\rangle)\ then\ \langle pattern\rangle$. For instance, Fig. 10 shows an invariant with an enabling condition to the left, so that the invariant is required to be satisfied only for

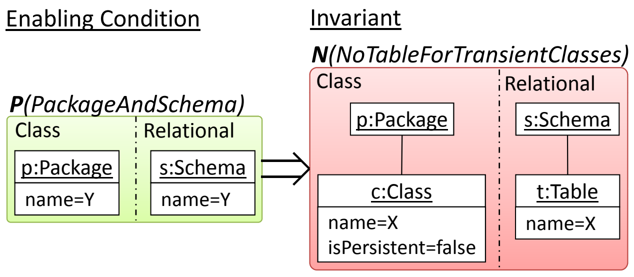


Fig. 10 Invariant with an enabling condition

packages for which there is an equally named schema. In such a case, the invariant states that the transient classes inside the packages should not have a corresponding table in the schema (because the invariant is negative). This pattern uses a non-constructive specification style, ensuring that a transformation implementation will not accidentally translate a non-persistent class into a table.

Fig. 11 shows to the left the scheme of an invariant with one enabling and one disabling condition, while the right part sketches its evaluation on a pair of models. In this case we look for all occurrences of the source object graph of the invariant *plus* the enabling condition, which in addition: (i) fulfill the expression EXP^{EN} of the enabling condition, (ii) fulfill the part of the invariant expression containing only source elements ($EXP|_{src}$), and (iii) for which no occurrence of the disabling condition (which might contain an expression EXP^{DS}) is found. Then, for each one of these occurrences, there should be an occurrence of the target object graph of the invariant satisfying the invariant expression. Note how enabling conditions permit including target elements in the pattern condition (i.e., in the *for all*).

The evaluation of invariants with enabling and disabling conditions is therefore as follows:

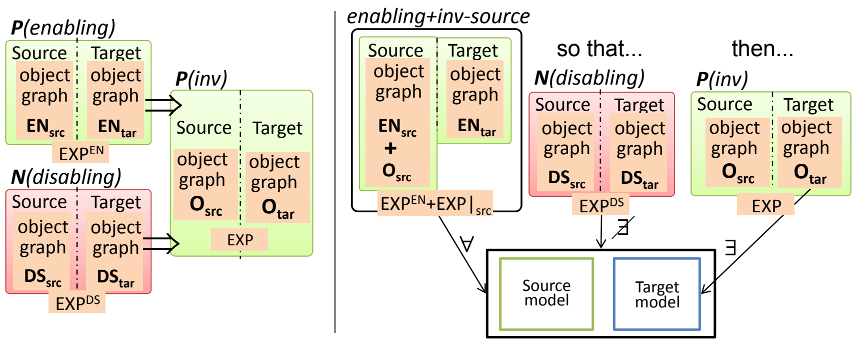

P(enabling)
Source object graph EN_{src}
Target object graph EN_{tar}
EXP^{EN}
P(inv)
Source object graph O_{src}
Target object graph O_{tar}
EXP
N(disabling)
Source object graph DS_{src}
Target object graph DS_{tar}
EXP^{DS}
enabling+inv-source
Source object graph EN_{src} + O_{src}
Target object graph EN_{tar}
EXP^{EN}+$EXP|_{src}$
so that...
N(disabling)
Source object graph DS_{src}
Target object graph DS_{tar}
EXP^{DS}
then...
P(inv)
Source object graph O_{src}
Target object graph O_{tar}
EXP
∀
∄
∃
Source model
Target model

Fig. 11 Scheme of the semantics of enabling and disabling conditions

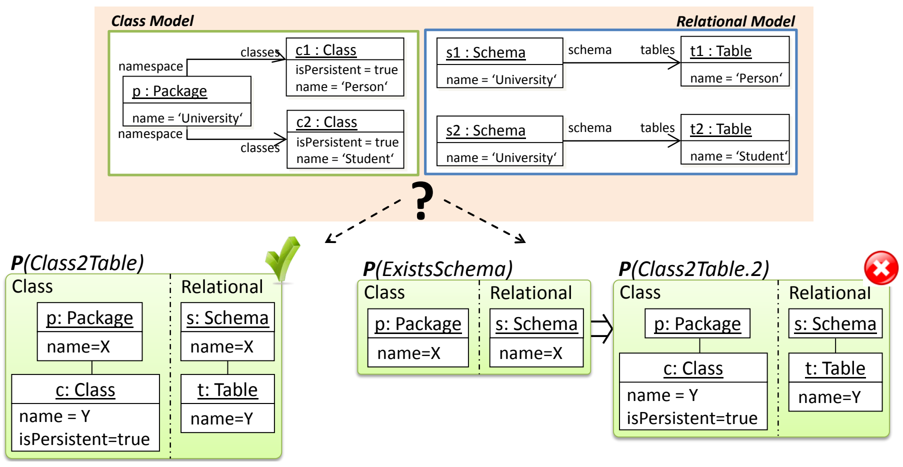


Fig. 12 Semantics of invariants with and without enabling condition

$$
\begin{aligned}
&\forall Occ(EN_{src} + O_{src}, EN_{tar})\ s.t.\\
&\quad [(EXP^{EN} + EXP|_{src})(Occ(EN_{src} + O_{src}, EN_{tar}))\wedge\\
&\quad\ \nexists Occ(DS_{src}, DS_{tar})\ s.t.\ EXP^{DS}(Occ(DS_{src}, DS_{tar})) \wedge ...]\\
&\quad\quad \exists Occ(O_{tar})\ s.t.\ EXP(Occ(O_{src}), Occ(O_{tar}))
\end{aligned}
$$

Fig. 12 illustrates how enabling conditions modify the semantics of a pattern, through an example of two syntactically similar invariants for classes, one declaring an enabling condition and the other not. The invariant in the lower left demands the existence of a schema and table for each persistent class in a package. The models shown above fulfill this, as the class model contains two occurrences of the source graph of the invariant (i.e., two classes), and for each one we find one schema in the relational model defining a table with same name as the class. In contrast, the models do not satisfy the invariant to the right. This is so as this invariant demands that for each occurrence of a persistent class, its package and equally named schema (this latter required by the enabling condition), a table with same name as the class exists. This is not true in this case as, for instance, if we take the occurrence given by objects `p`, `c1` and `s2`, there is no table named “Person” in `s2`.

Pre- and postconditions may have enabling and disabling conditions as well. As an example, Fig. 13 shows to the left the scheme of the semantic interpretation of a precondition with an enabling condition. In this case, for each occurrence of the enabling condition, we need to find an occurrence of the precondition. For the sake of illustration, the right part of the figure shows an example precondition demanding each persistent class to have at least one attribute.

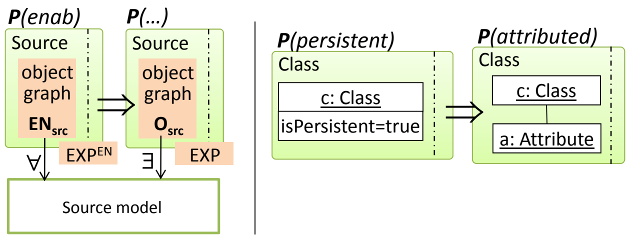


Fig. 13 Semantics of precondition with enabling condition (left). Example (right)

4.1.3 Sets

It is sometimes useful to formulate properties related to the number of times a certain structure can occur in a model. For this purpose, patterns can define variable sets of source and target elements (improving the expressive power compared to [22]). A set is depicted as a polygon with a name (see for example set `pclasses` in Fig. 14) and it represents the set of all occurrences of the structure enclosed in the polygon. Furthermore, sets may be nested and contain arbitrary structures. As an example, the left side of Fig. 14 shows an invariant making use of sets in the source and target. The invariant states that the number of persistent classes in a package (size of set `pclasses`) should be the same as the number of tables in the corresponding schema (size of set `tables`).

The center and right sides of Fig. 14 show the evaluation scheme of invariants with sets. The figure in the middle represents an invariant with two sets (`set1` in the source and `set2` in the target) and a constraint expression EXP that includes both sets. A pair of models satisfies such an invariant if for each occurrence of the source object graph, there is an occurrence of the target object graph that satisfies the constraint expression. Such an expression may make use of the sets `set1` and `set2` of all occurrences of the object graphs O_{set1} and O_{set2}:

$$\begin{aligned}
&\forall Occ(O_{src})\ s.t.\ EXP|_{src}(O_{src}, Set\ of\ all\ Occ(O_{set1}))\\
&\quad \exists Occ(O_{tar})\ s.t.\\
&\qquad EXP(Occ(O_{src}), Occ(O_{tar}), Set\ of\ all\ Occ(O_{set1}), Set\ of\ all\ Occ(O_{set2}))
\end{aligned}$$

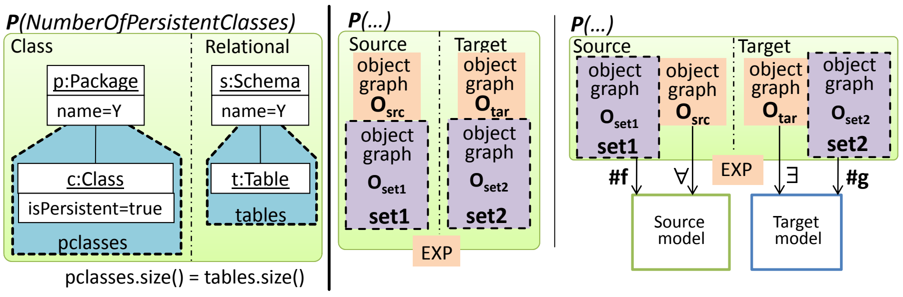


Fig. 14 Invariant with sets (left). Semantics of invariants with sets (center and right)

4.2 Reasoning with Patterns

The formal semantics of PaMoMo allows for reasoning on: (i) *metamodel coverage*, (ii) *redundancies*, (iii) *contradictions* and (iv) *pattern satisfaction* on contracts, as we detail next.

First, we can measure *metamodel coverage*, that is, we can identify the elements in the source and target metamodels that are used in a PaMoMo contract, as well as how they are used (i.e., in enabling or disabling conditions only, or in positive/negative patterns). This allows for a quick identification of underspecifications if, for instance, some element in the target metamodel is not used in any positive pattern. In the presented example in Figs. 5–14 all elements in both metamodels are used.

Second, we can investigate *redundancies* in contracts (cf. Table 1). A redundant pattern can be safely removed yielding a simpler, more compact contract with the same semantics as the original one but which can be more efficiently verified. For instance, if a positive pre- or postcondition is included in a "bigger" positive pre- or postcondition, the smaller one is redundant and can be removed. The reason is that whenever the bigger one is found, the smaller one will be found as well (and both need to be found). Similarly, if a negative pre- or postcondition is included in a "bigger" one, the bigger is redundant. Table 1 shows these two redundancy cases (first row), as well as other cases that we can identify for invariants (second row) and for the disabling conditions of a pattern (third row). For example, if a pattern has a disabling condition included in another one, then the "bigger" condition is redundant. Please note that the redundancy rules for invariants assume the forward interpretation of patterns; in the backward case the rules are the symmetric ones.

Table 1 Redundancies in PaMoMo contracts. P_i and N_i are a positive and a negative pattern without enabling or disabling conditions. Subindex *src* and *tar* refer to the source and target of a pattern.

Scope	**Rule**
Pre/postconditions	$P_1 \subseteq P_2 \Rightarrow P_1$ is redundant $N_1 \subseteq N_2 \Rightarrow N_2$ is redundant
Invariants	$P_{1,src} = P_{2,src}$ and $P_{1,tar} \subseteq P_{2,tar} \Rightarrow P_1$ is redundant $P_{1,tar} = P_{2,tar}$ and $P_{1,src} \subseteq P_{2,src} \Rightarrow P_2$ is redundant $N_{1,src} = N_{2,src}$ and $N_{1,tar} \subseteq N_{2,tar} \Rightarrow N_2$ is redundant $N_{1,tar} = N_{2,tar}$ and $N_{1,src} \subseteq N_{2,src} \Rightarrow N_2$ is redundant
Disabling conditions of a pattern	$disabling_1 \subseteq disabling_2 \Rightarrow disabling_2$ is redundant

Third, we can statically investigate *contradictions* preventing the satisfaction of a contract by any pair of models (cf. Table 2). For example, there is a contradiction if a negative pre- or postcondition is included in a positive pre- or postcondition. The reason is that the satisfaction of the positive precondition requires finding an occurrence in the source model, but this means that we will find an occurrence of the negative precondition as well. This conflict corresponds to the first two rows in Table 2. The third row in the table shows another contradiction that may arise if a negative postcondition is included in the target of a positive invariant. In this case, the invariant and the post-

Table 2 Contradictions in PaMoMo contracts. *Pre*, *Pos* and *I* refer to a precondition, postcondition and invariant without enabling or disabling conditions. Prefix *P* and *N* mean positive or negative. Subindex *src* and *tar* refer to the source and target of a pattern.

Scope	**Contradiction**
Pre/postconditions	$NPre \subseteq PPre \Rightarrow$ contract is unsatisfiable $NPos \subseteq PPos \Rightarrow$ contract is unsatisfiable $NPos \subseteq PI_{tar} \Rightarrow$ contract is potentially unsatisfiable
Invariants	$NI_{src} = PI_{src}$ and $NI_{tar} \subseteq PI_{tar} \Rightarrow$ contract is potentially unsatisfiable

condition cannot be simultaneously satisfied whenever we find an occurrence of the source part of the invariant in the source model, and only if the source model does not contain the source part of the invariant the contract may be satisfied (by vacuous satisfaction of the invariant). The same situation arises if two invariants have the same source, one is positive and the other negative, and the target of the negative one is included in the target of the positive one (last row in the table).

Finally, we can reason on the *satisfaction* of patterns in order to detect potential errors in a contract and report a warning. For instance, consider a negative precondition that is included in the source part of an invariant or in one of its enabling conditions. In this case there is no contradiction, but if the negative precondition holds, then the invariant will also hold vacuously because it will never be enabled. If the precondition does not hold, then the invariant can be satisfied or not (depending on whether its main pattern is found in the models) but nevertheless the whole contract will not hold. Thus, this situation usually indicates an error in the specification. Table 3 gathers different warnings for PaMoMo contracts concerning satisfiability.

Table 3 Potential errors in PaMoMo contracts concerning satisfiability. *Pre*, *Pos* and *I* refer to a precondition, postcondition and invariant. Prefix *P* and *N* mean positive or negative (the absence of prefix means "in both cases"). Subindex *src* and *tar* refer to the source and target of a pattern.

Scope	**Warning**
Pre/postconditions	$NPre \subseteq I_{src} \Rightarrow$ if $NPre$ holds, I vacuously holds $NPre \subseteq I_{enabling} \Rightarrow$ if $NPre$ holds, I vacuously holds $NPos \subseteq NI_{tar} \Rightarrow$ if $NPos$ holds, NI holds $NPos \subseteq I_{enabling} \Rightarrow$ if $NPos$ holds, I vacuously holds
Enabling/disabling conditions of a pattern	*disabling* $\subseteq$ *enabling* $\Rightarrow$ pattern vacuously holds

As an example of this kind of reasoning, Fig. 15 shows on top a negative precondition discarding the transformation of models where some package contains duplicated classes. The invariant below, specified by a different designer, deals with the transformation of equally named classes inside a package, which should be transformed into a single table containing columns for the attributes of the classes. Thus, the second invariant is useless because it can only be satisfied (in a non-vacuous way) if the input model has duplicated classes, but this is forbidden by the negative precondition. This situation, which corresponds to the first row in Table 3 (i.e., $NPre \subseteq I_{src}$), gives rise to a warning.

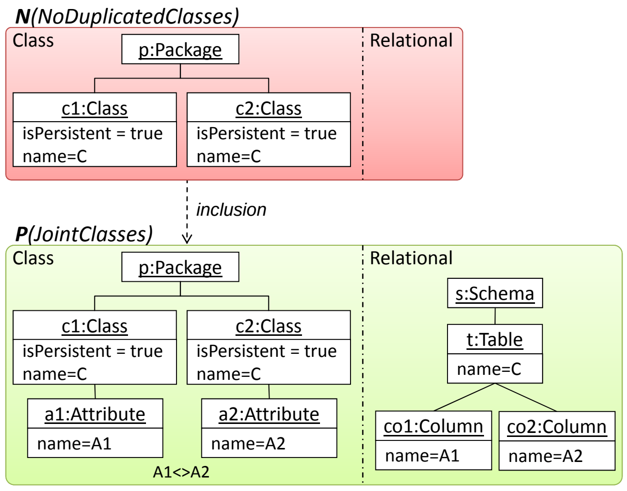


Fig. 15 Potential error: disabled invariant due to negative precondition

5 Implementing Model Transformations with QVT-Relations

After the designer has specified the transformation requirements in terms of contracts (cf. step 1 in Fig. 4), the developer may start implementing the model transformation (cf. step 2 in Fig. 4). Although any arbitrary transformation language might be chosen for this task, we employ QVT-Relations in our running example. This is since we will also use QVT-Relations to automatically verify the specified contracts (cf. Section 6), and thus, the reader is not confronted with many different languages.

QVT-Relations (QVT-R in short) is a declarative model transformation language standardized by the Object Management Group (OMG) [43]. It allows for several execution scenarios, like *model transformation* (i.e., generating a new target model from an existing source model), *model synchronization* (i.e., synchronizing two existing models) and *consistency checking* (i.e., checking the synchronization of two existing models without enforcing it).

With QVT-R, a transformation is specified as a set of relations that must hold between a set of models, called candidate models. Each relation defines local constraints to be satisfied by the candidate models, and has two or more domains. Domains are described by object graph patterns, and have a flag to indicate whether they are `checkonly` or `enforce`. The models of a domain marked as `enforce` may be modified to satisfy the relation. In contrast, the models of a domain marked as `checkonly` are just inspected to check if the relation holds for the candidate models, resulting in reported errors only. Thus, in order to realize a transformation scenario, the target domain must be marked as `enforce` to allow the creation of a new target model, and the transformation must be executed in the direction of this domain. In our example transformation, we aim at generating a new target model from an existing source model, and hence the domain `class` is marked as `checkonly` whereas the domain `rel` is marked as `enforce`.

Fig. 16 shows a first version of the QVT-R implementation for the running example. This transformation comprises two candidate models `class` and `rel` (cf. line 2) representing a model conforming to the `Class` metamodel and a model conforming to the `Relational` metamodel, respectively. The transformation contains five relations, namely `PackageToSchema`, `ClassToTable`, `AttributeToColumn`, `PrimitiveAttributeToColumn` and `SuperAttributeToColumn`. Relations may be *top-level* or not, which is indicated with the keyword `top`. The execution of a transformation requires that all its top-level relations hold, whereas the non-top level ones only need to hold when they are invoked directly or indirectly from top-level relations. A relation holds if for each binding of the objects in the source graph pattern (in the source model), there exists a valid binding of the target pattern objects (in the target model).

In the example, assuming that the execution starts with the top relation `ClassToTable` (cf. line 16), then it is required that for each persistent class `c` contained in a package `p`, a table `t` contained in a schema `s` exists. Furthermore, the class `c` and the table `t` must be equally named, which is enforced by using a common variable `cn`.

In addition, relations may declare *when* and *where* clauses containing OCL expressions as well as relation invocation expressions. *When* clauses express preconditions under which the relation needs to hold. They usually refer to other relations, to which they pass a number of parameters that appear as variables in the current relation. For instance, the relation `ClassToTable` is only required to hold if the relation `PackageToSchema` holds, as this latter relation appears in the *when* clause of `ClassToTable` (cf. line 28). *Where* clauses are used to specify relation postconditions (i.e., if the current relation holds then the *where* clause should hold) and may also include references to other relations. For instance, `ClassToTable` requires the relation `AttributeToColumn` to hold in its *where* clause (cf. line 31). This second relation delegates the transformation of attributes to the relations `PrimitiveAttributeToColumn` and `SuperAttributeToColumn` in its *where* clause (cf. lines 39 and 40). The relation `PrimitiveAttributeToColumn` transforms the attributes of a class `c` into equally named and typed columns of the corresponding table. Finally, the relation `SuperAttributeToColumn` deals with inherited attributes by recursively calling itself (cf. line 68).

As the attentive reader might have already spotted, by the recursive call in the *where* clause of the `SuperAttributeToColumn` relation, all super classes of a given class are visited, but without producing additional columns for inherited attributes. In Section 7, we show how this error is detected by using the previously presented contract and how it can be fixed. For this purpose, the next section shows how to use the consistency checking mechanisms of QVT-R to verify PaMoMo contracts.

```
transformation ClassToRel
 (class : Class ; rel : Relational ){

 // map each package to a schema
 top relation PackageToSchema {
  pn: String ;
  checkonly domain class p: Package {
   name =pn
  };
   enforce domain rel s: Schema {
   name =pn
  };
 }

 // map each persistent class to a table
 top relation ClassToTable {
  cn: String ;
  checkonly domain class c: Class {
   namespace =p: Package {},
   isPersistent =true ,
   name =cn
  };
  enforce domain rel t: Table {
   schema =s: Schema {},
   name =cn
  };
  when {
   PackageToSchema (p, s);
  }
  where {
   AttributeToColumn (c, t);
  }
 }
 // map each attribute to a column
 relation AttributeToColumn {
  checkonly domain class c: Class {};
  enforce domain rel t: Table {};
  where {
   PrimitiveAttributeToColumn (c, t);
   SuperAttributeToColumn (c, t);
  }
 }

 // map each attribute to a column
 relation PrimitiveAttributeToColumn {
  an , tn: String ;
  checkonly domain class c: Class {
   attributes =a: Attribute {
    name =an,
    type =tn
   }
  };
  enforce domain rel t: Table {
   columns =cl: Column {
    name =an ,
    type =tn
   }
  };
 }

 // map inherited attributes
 relation SuperAttributeToColumn {
  checkonly domain class c: Class {
   parents=sc: Class {}
  };
  enforce domain rel t: Table {};
  where {
   SuperAttributeToColumn (sc , t);
  }
 }
}
```

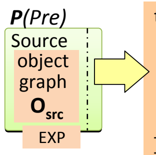
Fig. 16 Class2Relational transformation implemented in QVT-R

6 Operationalizing Contracts: From PaMoMo to QVT-Relations

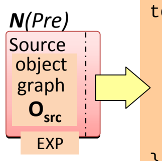
In order to use PaMoMo contracts as oracles, they have to be made operational. For this purpose, we translate the contracts into checkonly QVT-R transformations and check if they hold for certain models, according to the semantics shown in Section 4. In case a certain relation does not hold, the QVT engine provides information on which contract failed due to which bindings (i.e., bound objects, values and links). Our approach generates three QVT transformations: one containing the generated code for the preconditions, another one for the invariants, and the last one for the postconditions. In the following we detail each one of them by providing a schematic template of the generated code and a concrete example.

6.1 Compilation of Preconditions and Postconditions

Compilation scheme of preconditions. Preconditions have empty the target compartment. However, in QVT-R, all transformations must have at least two domains. Therefore, in the case of pre- and postconditions, we generate



```
top relation ⟨Pre⟩ {

  domain Source1 ⟨Root of Object Graph O_src⟩
  checkonly domain Source2 ⟨Object Graph O_src⟩

  where { ⟨obj-identity-inequality⟩; ⟨EXP⟩; }
}
```



```
top relation ⟨Pre⟩ {

  domain Source1 ⟨Object Graph O_src⟩
  checkonly domain Source2 ⟨Root of Object Graph O_src⟩

  when  { ⟨obj-identity-inequality⟩; ⟨EXP⟩; }
  where { false; }
}
```

Fig. 17 Compilation scheme for preconditions

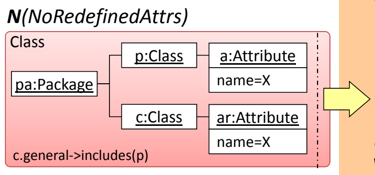
transformations with two domains conforming to the same metamodel, which are actually bound to the same model. Fig. 17 shows the compilation scheme for positive and negative preconditions. In both cases, we produce one top relation with two domains (named `Source1` and `Source2` in the figure) bound to the same metamodel.

If we execute the resulting transformation in check-only mode in the direction `Source1`→`Source2`, for each occurrence of the source of each top relation, the engine has to find an occurrence of the target of the relation to consider that the relation holds. Therefore, for positive preconditions, we add in `Source1` one element that we will always need to find (the root node of the precondition's object graph), and in `Source2` the full object graph. Furthermore, we include in the *where* clause inequalities ensuring that two objects with compatible type cannot be bound to the same object in the model, as well as the OCL constraint expression `EXP` of the precondition.

Regarding negative preconditions, they demand the absence of an object graph. Therefore, in this case, the object graph is added in the `Source1` domain, and the OCL constraint is included in the *when* clause. Moreover, as a negative precondition has to fail whenever the object graph is found, we add *false* to the *where* clause of the relation. Thus, finding the object graph in the source domain makes the relation fail due to the *where* clause.

Example. Fig. 18 shows a negative precondition taken from Fig. 8 and the generated QVT-R code. The source object graph of the negative precondition is compiled as the object graph for the `Source1` domain, whereas the `Source2` domain includes only the root node of this graph. In addition, three constraints are added to the *when* clause. The first two check that different objects in the relation are bound to different objects in the model. This is checked by inequalities in the identifiers of objects with same type. The third constraint, taken directly from the precondition, checks if the class `p` is a superclass of class `c`. Finally, the *where* clause includes the *false* statement, to make the relation fail in case a match for the source graph is found in the model.

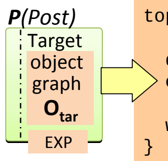
Compilation scheme of postconditions. Fig. 19 shows the scheme of the compilation of positive and negative postconditions. Positive postcondi-



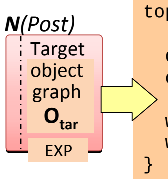
```
transformation checkPre(Source1:uml; Source2:uml){
 top relation NoRedefinedAttrs{
   X : String;
   domain Source1 pa : Package{
      classes = p : Class{
         attribute = a : Attribute{ name=X }
       },
      classes = c : Class{
          attribute = ar : Attribute{ name=X }
       }
   };
   checkonly domain Source2 pa2 : Package{};
   when{
      c<>p;
      ar<>a;
      c.general->includes(p);
   }
   where{ false; }
 }
}
```

Fig. 18 Compiling a negative precondition into QVT-R

tions demand an occurrence of the target object graph, while negative post-conditions are satisfied if there is no occurrence of the target object graph. Thus, the code generated from postconditions is similar to the one generated from preconditions but acting on the target metamodel.



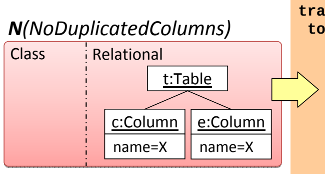
```
top relation ⟨Post⟩ {

  domain Target1 ⟨Root of Object Graph O_tar⟩
  checkonly domain Target2 ⟨Object Graph O_tar⟩

  where { ⟨obj-identity-inequality⟩; ⟨EXP⟩; }
}
```



```
top relation ⟨Post⟩ {

  domain Target1 ⟨Object Graph O_tar⟩
  checkonly domain Target2 ⟨Root of Object Graph O_tar⟩

  when  { ⟨obj-identity-inequality⟩; ⟨EXP⟩; }
  where { false; }
}
```

Fig. 19 Compilation scheme for postconditions

Example. Fig. 20 depicts the negative postcondition shown in Fig. 8 and its compilation into QVT-R. It can be seen that the resulting code is analogous to the code produced for the negative precondition example in Fig. 18.

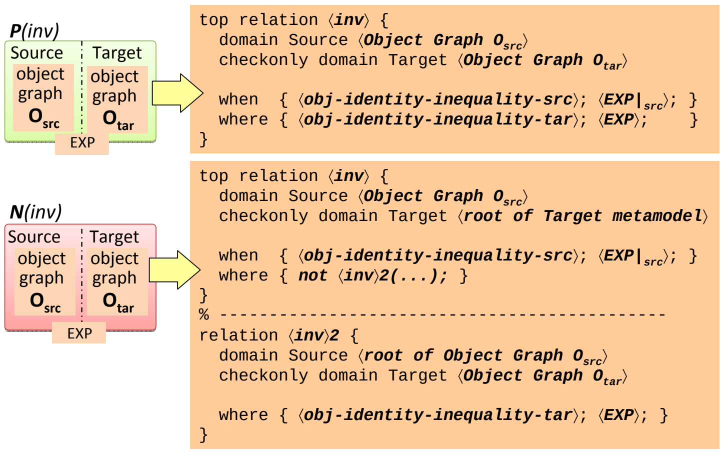


```
transformation checkPost(Target1:rdbms; Target2:rdbms){
 top relation NoDuplicatedColumns {
   X : String;
   domain Target1 t : Table {
      columns = c:Column { name=X },
      columns = e:Column { name=X }
   };
   checkonly domain Target2 t2 : Table {};
   when  { c <> e; }
   where { false; }
 }
}
```

Fig. 20 Compiling a negative postcondition into QVT-R

6.2 Compilation of Invariants

Compilation scheme. Fig. 21 shows the scheme of the compilation of positive and negative invariants. The scheme for positive invariants is similar to the one for preconditions and postconditions, but now the two domains are typed on different metamodels and contain different object graphs. Moreover, the *when* clause includes the terms of the OCL invariant expression containing only elements of the source graph, whereas the remaining terms of the expression are added to the *where* clause.



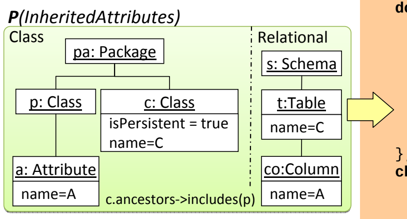
```
top relation ⟨inv⟩ {
  domain Source ⟨Object Graph O_src⟩
  checkonly domain Target ⟨Object Graph O_tar⟩

  when  { ⟨obj-identity-inequality-src⟩; ⟨EXP|_src⟩; }
  where { ⟨obj-identity-inequality-tar⟩; ⟨EXP⟩;      }
}
```

```
top relation ⟨inv⟩ {
  domain Source ⟨Object Graph O_src⟩
  checkonly domain Target ⟨root of Target metamodel⟩

  when  { ⟨obj-identity-inequality-src⟩; ⟨EXP|_src⟩; }
  where { not ⟨inv⟩2(...); }
}
% ---------------------------------------------
relation ⟨inv⟩2 {
  domain Source ⟨root of Object Graph O_src⟩
  checkonly domain Target ⟨Object Graph O_tar⟩

  where { ⟨obj-identity-inequality-tar⟩; ⟨EXP⟩; }
}
```

Fig. 21 Compilation scheme for invariants

As a difference from the previous compilations, negative invariants are split into two relations: the first one is top and looks for occurrences of the source, and the second one is non-top and looks for occurrences of the target when it is invoked from the *where* clause of the top relation. In this way, the top relation checks that for each occurrence of the source graph, there is no occurrence of the target graph (this latter checked by invoking the non-top relation in the *where* section, negated). Note that generating a single relation with a *false* statement in the *where* section, as we did for negative pre- and postconditions (cf. Figs. 17 and 19), is not enough in this case. The reason is that such a relation fails if it does not find the complete target graph, however the relation should fail only if it does find both the source and target graphs.

Example. Fig. 22 shows the compilation of the positive invariant modeling requirement 4 in Fig. 7. The generated relation has one domain for the source object graph and another domain for the target object graph. Its *when* clause includes an inequality to avoid binding the two classes `p` and `c` to the same object in the model, as well as the OCL constraint in the invariant as it only includes source objects. An example for the compilation of negative invariants is illustrated in the following subsection.

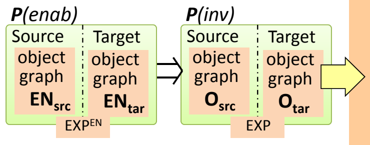


```
transformation checkInv(Source:uml; Target:rel){
 top relation InheritedAttributes {
   A, C : String;
   domain Source pa : Package {
      classes = p:Class {
         attributes = a:Attribute { name = A }
      },
      classes = c:Class {
        isPersistent = true,
        name = C
      }
   };
   checkonly domain Target s : Schema {
      tables = t:Table {
         name = C,
         columns = co:Column { name = A }
      }
   };
   when { p<>c; c.ancestors->includes(p);}
 }
}
```

Fig. 22 Compiling a positive invariant into QVT-R

6.3 Compilation of Enabling and Disabling Conditions

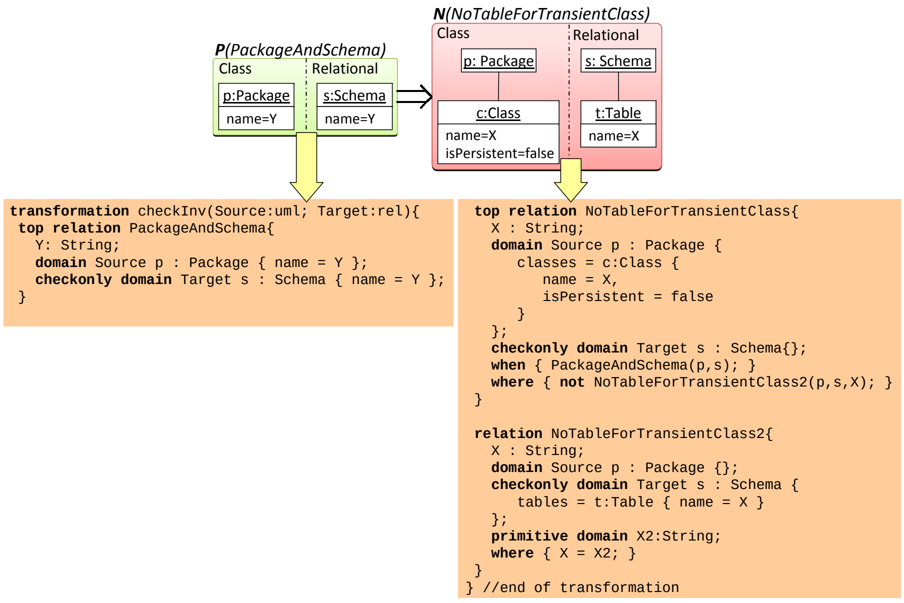
Compilation scheme. Enabling conditions are translated into top relations, which are checked in the *when* clause of the relation derived from the pattern they constrain. In this way, if the relation derived from the enabling condition does not hold, then the relation derived from the pattern vacuously holds. This compilation scheme is shown in Fig. 23. For disabling conditions the scheme is the same, but they are invoked in the *when* clause preceded by "not". If a pattern contains several disabling conditions, their invocations are concatenated with a logical "and".



```
top relation ⟨inv⟩ {
  domain Source ⟨Object Graph O_src⟩
  checkonly domain Target ⟨Object Graph O_tar⟩

  when  { ⟨obj-identity-inequality-src⟩;
          ⟨EXP|_src⟩;
          ⟨enab⟩(…); }
  where { ⟨obj-identity-inequality-tar⟩;
          ⟨EXP⟩;      }
}
%-----------------------------------------
top relation ⟨enab⟩ {
  domain Source ⟨Object Graph EN_src⟩
  checkonly domain Target ⟨Object Graph EN_tar⟩

  when  { ⟨obj-identity-inequality-src⟩;
          ⟨EXP^EN|_src⟩; }
  where { ⟨obj-identity-inequality-tar⟩;
          ⟨EXP^EN⟩;    }
}
```

Fig. 23 Compilation scheme for enabling conditions

Example. Fig. 24 shows the code generated for the negative invariant of Fig. 10, which has an enabling condition. In particular, the relations `NoTableFor-TransientClass` and `NoTableForTransientClass2` are generated from the negative invariant, and `PackageAndSchema` from the enabling condition. Hence, top relation `NoTableForTransientClass` only needs to hold for a particular



```
transformation checkInv(Source:uml; Target:rel){
 top relation PackageAndSchema{
   Y: String;
   domain Source p : Package { name = Y };
   checkonly domain Target s : Schema { name = Y };
 }
```

```
 top relation NoTableForTransientClass{
   X : String;
   domain Source p : Package {
      classes = c:Class {
         name = X,
         isPersistent = false
      }
   };
   checkonly domain Target s : Schema{};
   when { PackageAndSchema(p,s); }
   where { not NoTableForTransientClass2(p,s,X); }
 }

 relation NoTableForTransientClass2{
   X : String;
   domain Source p : Package {};
   checkonly domain Target s : Schema {
      tables = t:Table { name = X }
   };
   primitive domain X2:String;
   where { X = X2; }
 }
} //end of transformation
```

Fig. 24 Compiling an enabling condition for a negative invariant into QVT-R

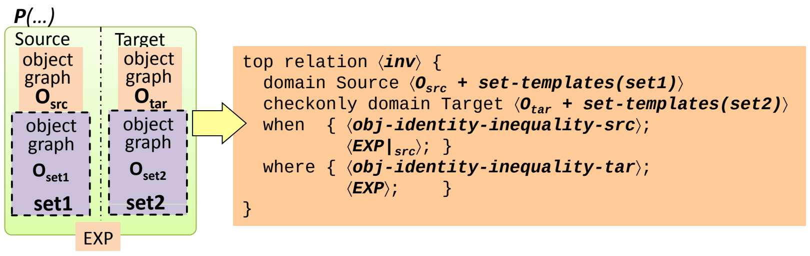
`Package` and `Schema` when they satisfy the relation `PackageAndSchema`, which is checked in the *when* clause.

In this example, the relation `NoTableForTransientClass` invokes `NoTable-ForTransientClass2` passing the string variable `X` as a parameter, which has to be defined as a primitive domain in the invoked relation. Moreover, due to a limitation of the QVT-R engine that we use (ModelMorf), which only supports relations with two domains, we have to tweak the compilation of enabling conditions containing more than one object in the source or target. This is so as any invocation to a relation must receive exactly two objects as parameters, plus any number of primitive values. Thus, if the enabling condition contains several objects in the source or the target, all should be passed in the invocation, which is not allowed. We have solved this problem by passing the object identifiers (which have primitive type, and can therefore be passed as primitive domains) instead of the objects themselves.

6.4 Compilation of Sets

Compilation scheme. QVT-R allows matching for collections of objects (sets, bags or sequences) using so-called *collection templates*. The ModelMorf QVT engine provides two kinds of collection templates: (i) *enumerations* for the extensional definition of sets, and (ii) *comprehensions* for its intensional definition. Enumerations match for a certain number of members in a collection. For instance, `classes = pclasses : Set(Class) {c1, c2 ++ _ }` matches for two classes in the reference `classes`. The underscore is a wildcard that matches for the rest of the collection. Comprehensions allow matching members in a collection using a condition. For instance, `classes = pclasses : Set(Class) {} {pclasses->forall (c | c.isPersistent)}` matches all persistent classes in the reference `classes`.

As Fig. 25 shows, sets in PaMoMo patterns are compiled into collection templates. We generate enumerations if the elements in the set are not constrained by any condition, and comprehensions otherwise. As before, the OCL expressions using only source variables and source set variables are included in the *when* clause, whereas the rest are included in the *where* clause.

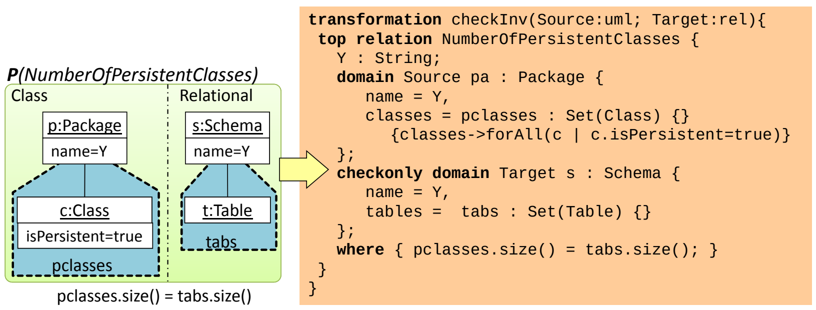


```
top relation ⟨inv⟩ {
  domain Source ⟨O_src + set-templates(set1)⟩
  checkonly domain Target ⟨O_tar + set-templates(set2)⟩
  when  { ⟨obj-identity-inequality-src⟩;
          ⟨EXP|_src⟩; }
  where { ⟨obj-identity-inequality-tar⟩;
          ⟨EXP⟩;    }
}
```

Fig. 25 Compilation scheme for sets

Example. Fig. 26 lists the code generated from the invariant with sets shown in Fig. 14. The set `pclasses` is translated into a comprehension because it contains a condition matching for persistent classes only (`isPersistent = true`). In contrast, the set `tabs` is compiled into a simple enumeration. The OCL expression is added to the *where* section of the relation because it relates set variables of the source and target. This expression fails if the number of persistent classes is not equal to the number of tables.

If a set contains an arbitrary graph having more than one element, then we generate one additional relation looking for occurrences of this graph structure. This relation is used to filter which elements should be added to the collection (i.e., only those making the relation hold).



```
transformation checkInv(Source:uml; Target:rel){
 top relation NumberOfPersistentClasses {
   Y : String;
   domain Source pa : Package {
      name = Y,
      classes = pclasses : Set(Class) {}
         {classes->forAll(c | c.isPersistent=true)}
   };
   checkonly domain Target s : Schema {
      name = Y,
      tables =  tabs : Set(Table) {}
   };
   where { pclasses.size() = tabs.size(); }
 }
}
```

Fig. 26 Compiling a positive invariant with sets into QVT-R

6.5 Summary of the Compilation

Table 4 summarizes the compilation of PaMoMo contracts into QVT-R code. We can observe that PaMoMo allows for a more compact specification of contracts than the direct use of QVT-R. Its graphical nature, the availability of different kinds of patterns (positive and negative invariants, pre- and post-conditions) and its features (enabling/disabling conditions, sets) make it less complex than the equivalent QVT-R code. In particular, patterns are especially useful to express negative information and large, complex graphical structures in a concise way. In these cases, the QVT-R code equivalent to the visual PaMoMo pattern is more intricate (e.g., see Fig. 24), as a negative invariant needs to be split in two QVT relations, and one additional relation needs to be generated for each enabling or disabling condition. The higher conciseness of PaMoMo for this task is natural, as its aim is specifying transformation properties, while QVT-R is a language to implement M2M transformations.

Table 4 Summary of PaMoMo-to-QVT compilation

PaMoMo concept	**QVT-R representation**
P(Pre/Post)	1 relation with pseudo domain
N(Pre/Post)	1 relation with pseudo domain + *false* in *where* clause
P(Inv)	1 relation
N(Inv)	2 relations + negated call to *relation2* from *where* of *relation1*
Enabling/Disabling	1 relation + (negated) call from *when* of relation produced for constrained pattern
Set	collection template

7 The PaCo-Checker Tool

After the transformation logic has been implemented, it needs to be verified to check whether it satisfies the requirements specified by the contract. For this purpose, we have developed an EMF-based tool [14] called PaCo-Checker that automates this process and enables the visual specification of contracts. Fig. 27 shows its architecture, which consists of three main components: a visual editor to build the patterns (label 1), a chain of transformations from the patterns to QVT-R abstract syntax and from this to QVT-R concrete textual syntax (label 2), and a verification editor to configure the patterns to be checked on a particular transformation using certain source and target models (label 3). The rest of this section provides an overview on the needed steps for the verification process and gives additional details of the tool components.

Prerequisites. We assume the existence of the source and target metamodels, as these are necessary to specify the contracts and implement the transformation. In addition, for the verification process, we need a suitable set of input models conforming to the source metamodel. Such input models

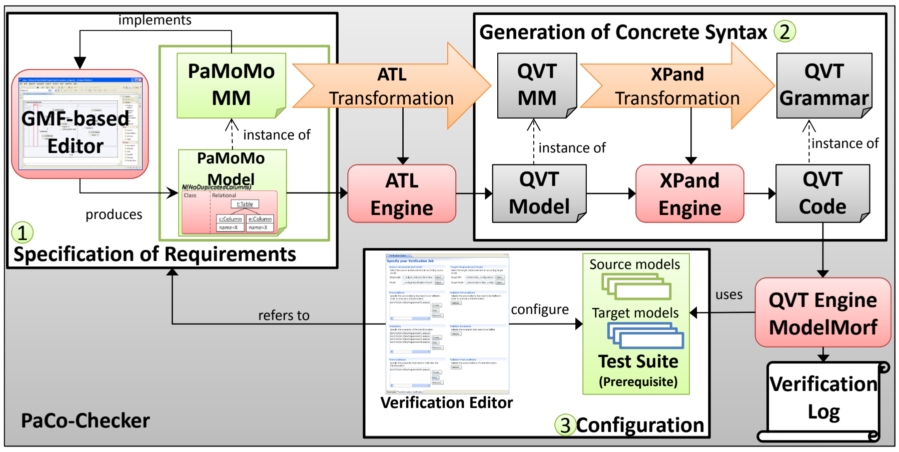


Fig. 27 Overview of the architecture of PaCo-Checker

can be manually created, which however is a tedious and error-prone task. In our experience, this manual creation often leads to small input models that only cover parts of the metamodel. Alternatively, there are available mechanisms that automatically synthesize a large number of different input models [9,17,47] ensuring a certain level of metamodel coverage. We assume the existence of such set of input models as well, since their generation is out of scope of this paper.

Step 1: Formal specification of requirements with PaMoMo. In a first step, the transformation requirements have to be formally specified using PaMoMo. For this purpose, PaCo-Checker implements the PaMoMo metamodel using EMF (cf. (1) in Fig. 27) and provides a graphical concrete syntax supported by a GMF-based [21] editor that enables the visual specification of contracts (cf. Fig. 28). The source and target metamodels of the transformation have to be imported into the tool palette of the editor before starting modeling patterns. Then, the transformation designer can use the editor to specify preconditions, postconditions and invariants. Our current implementation only supports one type of pattern per contract, i.e., either preconditions, postconditions or invariants, whereby one contract results in one file. Therefore, if preconditions, postconditions and invariants should be used to verify a transformation, three different contract files are needed. Fig. 28 shows a screenshot of the editor begin used to define the invariant that models the requirement 4 of the running example. Instances of the classes from the source and target metamodels can be added to the appropriate compartments of the invariant, together with the attributes of such classes. The patterns can also include OCL expressions to specify conditions on the attributes, e.g., to check if the class is persistent (boolean attribute `isPersistent`) or more general conditions, e.g., if class `p` is a superclass of class `c`.

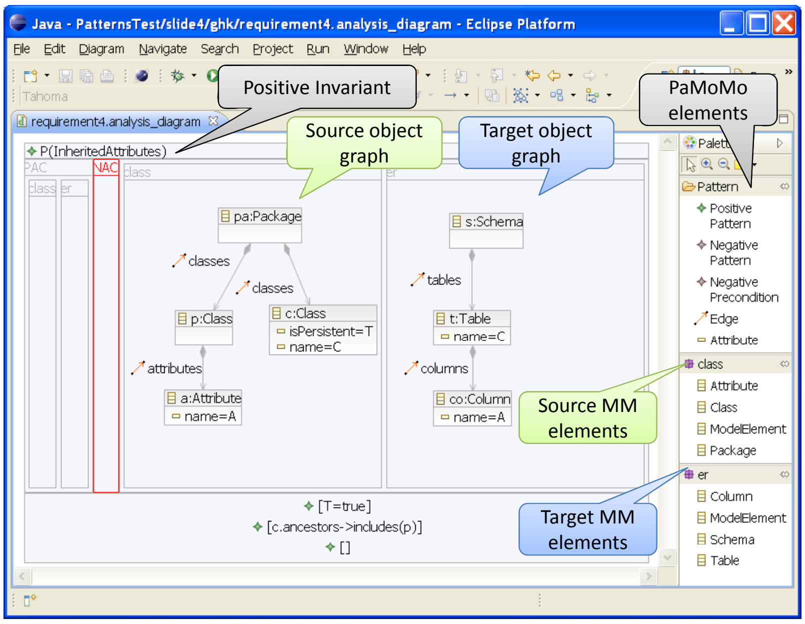


Fig. 28 Specification of invariant for *requirement 4* (cf. Fig. 7) with PaCo-Checker

Step 2: Specification of a verification job. Once the designer has specified the contracts, a *verification job* has to be configured (cf. (3) in Fig. 27). Such a job definition allows executing all specified preconditions, postconditions and invariants to achieve a comprehensive verification result. Fig. 29 shows a screenshot of the verification job for the running example. First, the source and target metamodels have to defined, which must be equal to those used for specifying the patterns. Furthermore, a source (test) input model is needed as well as the target model generated by the transformation under test. Then, the preconditions, postconditions and invariants which shall be checked for the transformation have to be selected. Thus, it is possible to reuse patterns to verify different transformations with overlapping requirements, e.g., if we have designed several transformations from the same source (target) metamodel, some of the preconditions (postconditions) may be reused.

Step 3: Execution of the verification job. Once specified, the verification job can be executed if no inconsistency between the patterns of the contract is reported by the reasoning component. In order to execute the job, an ATL transformation transforms the PaMoMo contract into a QVT model implementing the semantics of the contract (cf. (2) in Fig. 27). Since there is no execution engine available to execute QVT-R on the basis of its abstract syntax, we produce the QVT concrete textual syntax by means of a model-to-text transformation using XPand [52]. The resulting QVT-R code is finally executed by the ModelMorf QVT-R engine [49], which pro-

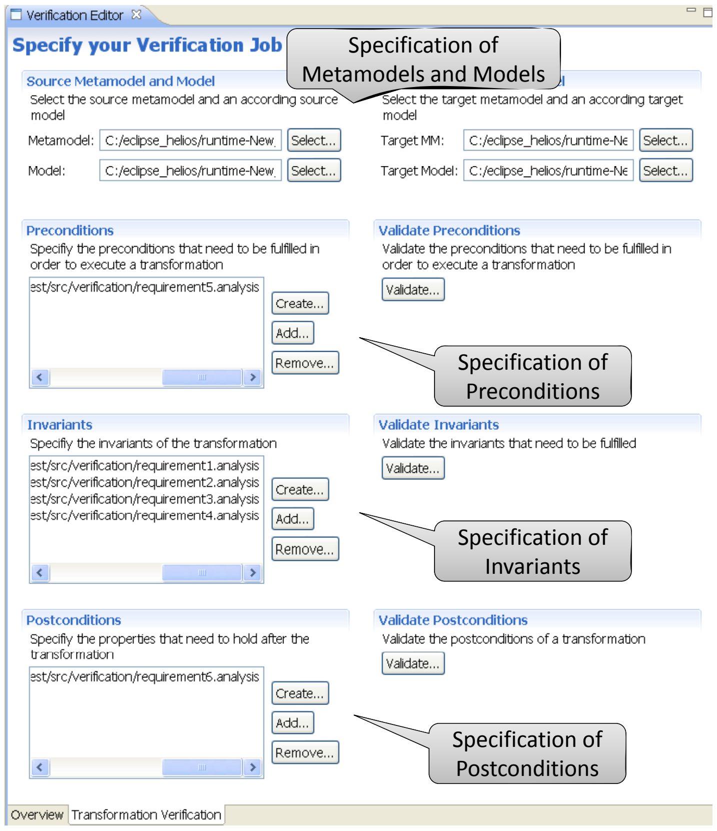
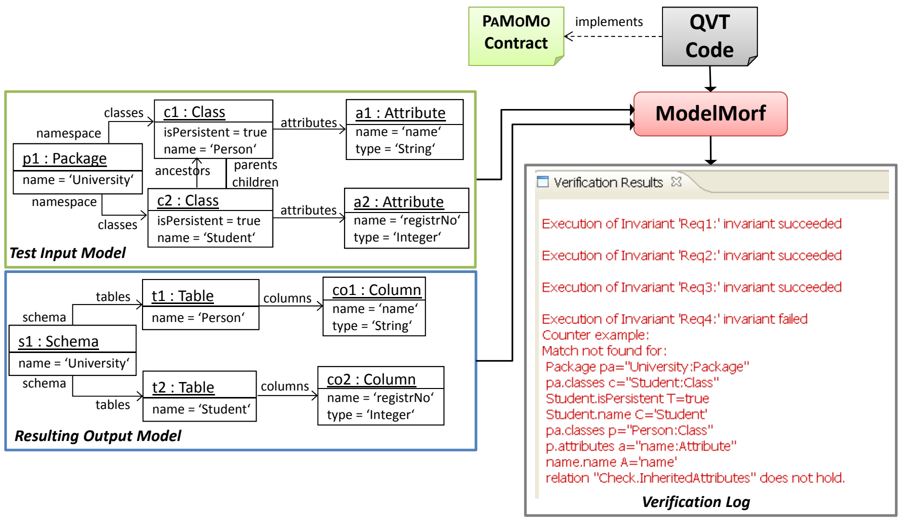

Verification Editor
Specify your Verification Job
Specification of Metamodels and Models
Source Metamodel and Model
Select the source metamodel and an according source model
Metamodel: C:/eclipse_helios/runtime-New_ Select...
Model: C:/eclipse_helios/runtime-New_ Select...
Select the target metamodel and an according target model
Target MM: C:/eclipse_helios/runtime-Ne Select...
Target Model: C:/eclipse_helios/runtime-Ne Select...
Preconditions
Specifiy the preconditions that need to be fulfilled in order to execute a transformation
est/src/verification/requirement5.analysis
Create...
Add...
Remove...
Validate Preconditions
Validate the preconditions that need to be fulfilled in order to execute a transformation
Validate...
Specification of Preconditions
Invariants
Specify the invariants of the transformation
est/src/verification/requirement1.analysis
est/src/verification/requirement2.analysis
est/src/verification/requirement3.analysis
est/src/verification/requirement4.analysis
Create...
Add...
Remove...
Validate Invariants
Validate the invariants that need to be fulfilled
Validate...
Specification of Invariants
Postconditions
Specify the properties that need to hold after the transformation
est/src/verification/requirement6.analysis
Create...
Add...
Remove...
Validate Postconditions
Validate the postconditions of a transformation
Validate...
Specification of Postconditions
Overview
Transformation Verification

Fig. 29 Definition of a verification job with PaCo-Checker

duces a verification log providing hints of any error in the transformation logic.

Step 4: Inspection of verification results. The execution of the verification job produces a verification log. Fig. 30 shows to the right the log generated for the running example, considering the input and output models to the left (the output model is produced by the transformation implementation). This log reports that these models satisfy requirements 1 - 3, but not requirement 4, which addresses the translation of inherited attributes. If we inspect the models, we realize that the transformation in Fig. 16 produces a schema `s1` which stems from the package `p1`, checked by the first invariant. The second invariant checks if persistent classes are translated into equally named tables, which is also true since two appropriate tables have been created. Furthermore, every direct attribute, i.e., `name` in case of class `c1` and `registrNo` in case of `c2`, has been correctly transformed into columns of the corresponding tables (as demanded by invariant 3). Nevertheless, invariant 4 fails because attribute `a1` in the superclass `c1` is not transformed into a column of the table generated from `c2` (i.e., table



```
Execution of Invariant 'Req1:' invariant succeeded

Execution of Invariant 'Req2:' invariant succeeded

Execution of Invariant 'Req3:' invariant succeeded

Execution of Invariant 'Req4:' invariant failed
Counter example:
Match not found for:
 Package pa="University:Package"
 pa.classes c="Student:Class"
 Student.isPersistent T=true
 Student.name C='Student'
 pa.classes p="Person:Class"
 p.attributes a="name:Attribute"
 name.name A='name'
 relation "Check.InheritedAttributes" does not hold.
```

Fig. 30 Verification results of requirements 1-4 for our running example

`t2` has no column `name`). Therefore, our transformation implementation in Fig. 16 does not handle appropriately the inherited attributes.

The transformation in Fig. 16 was implemented with the rationale that each relation addressed exactly one requirement. In this way, relation `SuperAttributeToColumn` handles requirement 4. However, when investigating the QVT code, it can be seen that we specified the wrong relation call in the *where* clause of relation `SuperAttributeToColumn`. In particular, by calling this relation, only the super classes are visited, but no column is created. We can solve this error by changing the *where* clause of relation `SuperAttributeToColumn` to call `AttributeToColumn` (cf. line 68 in Fig. 31) which takes care of, on the one hand, delegating the creation of additional columns, and on the other hand, traversing the super classes from bottom to top. Running again the updated transformation to produce the output model and subsequently verifying the contract shows that all invariants are satisfied.

```
61  // map inherited attributes
62  relation SuperAttributeToColumn {
63   checkonly domain class c: Class {
64    parents=sc: Class {}
65   };
66   enforce domain rel t: Table {};
67   where {
68    AttributeToColumn (sc , t);
69   }
70  }
71 }
```

Fig. 31 Corrected transformation code

8 Case Studies

In this section we illustrate the usefulness of contracts through several case studies in three application domains. The first one deals with the verification of the transformation from PaMoMo into QVT-R presented in this paper. The second one is concerned with the verification of a complex transformation from a process-interaction simulation language [15] in the area of performance evaluation into coloured Petri nets [25]. Finally, the third one presents an application of contracts for third-party transformations, in particular we tackle the generation of visual editors from GMF models. These case studies show the versatility and language independence of our approach by the automated verification of an ATL transformation, a QVT-R transformation, and the safe execution of a third party transformation (from which we do not have the source code). In each case, we stress the use of different features of PaMoMo.

8.1 Using PaMoMo to Verify its own Translation into QVT-Relations

In this section we show some patterns of the contract that helped us in verifying the transformation from PaMoMo into QVT-R. The metamodels of both languages are depicted in Fig. 32. With this example we want to stress that PaMoMo is independent from the language used to realize the transformations, since whereas our running example verified a QVT-R transformation, here the translation was implemented with ATL.

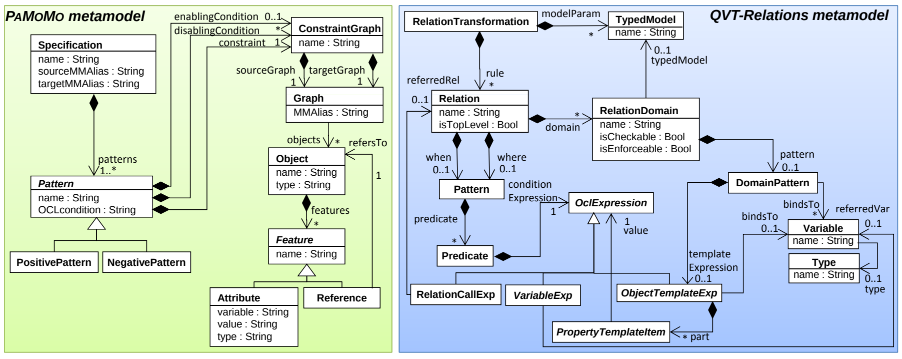


Fig. 32 PaMoMo (left) and QVT-R (right) metamodels

The contract for our transformation contains invariants and postconditions, but it does not contain preconditions because we handle the translation of all features in PaMoMo. As an example, Fig. 33 shows an invariant addressing the translation of pre and postconditions. These are patterns with either the source or target graphs empty (i.e., the size of the set of objects either in the source graph or the target graph is 0, as checked by the constraint expression). These patterns should be transformed into relations with two domains

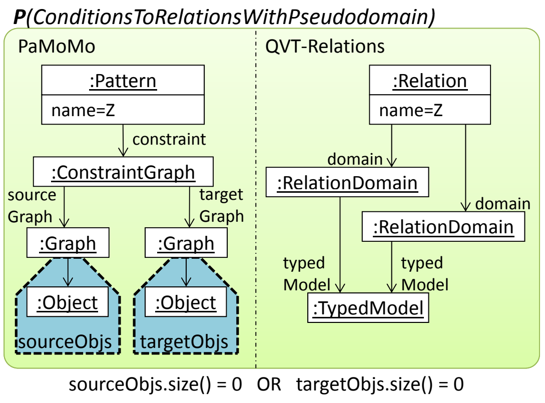


Fig. 33 A positive invariant for PaMoMo-to-QVT-R

(since this is required by the QVT-R metamodel) but referring to the same `TypedModel` instance, as shown by the target graph object.

Fig. 34 shows another invariant stating that positive patterns of any type without enabling or disabling conditions (checked by the two disabling conditions) are transformed into a unique relation. Thus, the generated relation cannot invoke other relations in its *when* clause (shown invariant) or *where* clause (checked by another similar invariant).

Finally, Fig. 35 shows two postconditions checking that all generated relation domains are `checkonly` (left), and that there are no chains of *when* relation invocations (right). Both are constraints of the models generated by our transformation.

8.2 From a Process-Interaction Language into Timed Coloured Petri Nets

If we are interested in modeling systems with the aim of simulating their performance, we can use a language in the process-interaction simulation style [15].

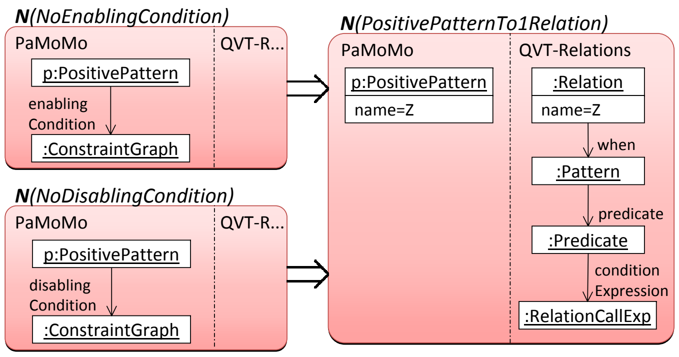


Fig. 34 A negative invariant for PaMoMo-to-QVT-R

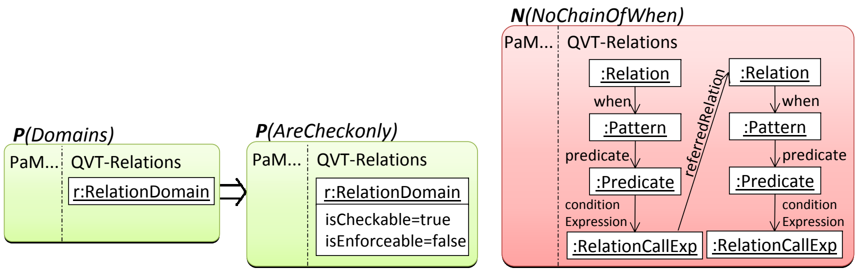


Fig. 35 Two postconditions for PaMoMo-to-QVT-R

In these kinds of languages, systems are modeled by processes made of interconnected blocks through which *transactions* flow.

Fig. 36 shows a process-interaction model. The two blocks to the left are *generators* of transactions. In particular, the upper left block produces a transaction of type 1 at each [10, 20] time steps, with a transaction length having a uniform probability between [120, 150]. Similarly, the lower left block produces a transaction of type 2 at each [12, 24] time steps, with a length having a uniform probability between [140, 180]. Both kinds of transactions arrive at the *advance* block (labeled "A"), which models a process with a delay given by a uniform probability in the interval [2, 5]. After this delay, transactions reach a *server* block with a parallelism of 3, meaning that the server can attend 3 transactions at the same time. Moreover, the server has a delay between [4, 5]. Then, a *type switch* block (labeled "type") selects the transactions depending on their type. Transactions of type 1 are routed into a server with parallelism 2, while transactions of type 2 are routed into a server with parallelism 3. Finally, transactions finish in a *terminate* block, which counts 1 each time a transaction arrives. Altogether, this model represents a client/server system that accepts two kinds of requests, processed in different servers.

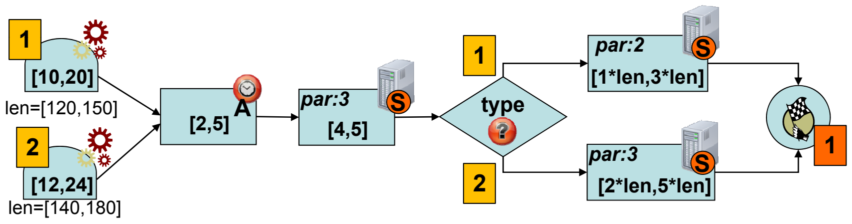


Fig. 36 A process-interaction model

Fig. 37 shows the metamodel for this process-interaction language. Thus, a `Simulation` model is made of `Blocks` and `Resources`. `Block` is an abstract class subclassified for each different kind of block.

In order to simulate and analyse process-interaction models, we have built a transformation of these models into Coloured Petri Nets (CPNs) [25], which allows using tools like CPNTools [26] for this task. CPNs are a kind of automaton made of two kinds of nodes: *places* and *transitions*. Transitions can

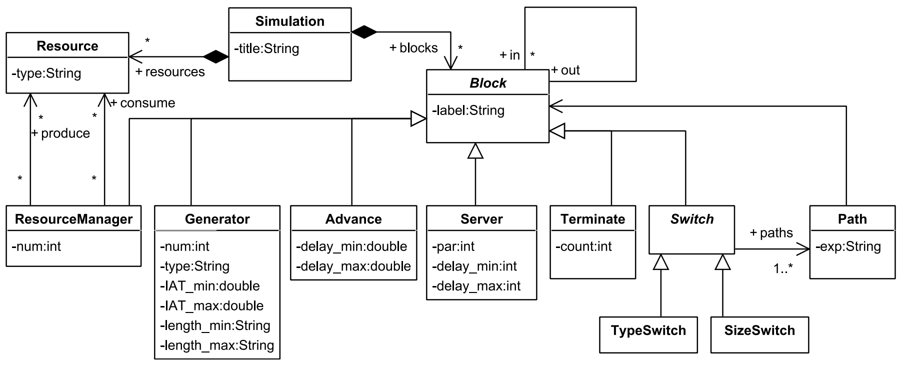


Fig. 37 Metamodel of the process-interaction language

be connected to places, and vice versa, using directed arcs. Places may contain *tokens*, and these may store data conformant to a given data type. Transitions are the active elements of the system. Whenever all its incoming places have at least one token, the transition is *enabled* and *may* fire, removing one token from each input place and adding one token to each output place. Arcs are decorated with inscriptions that select tokens from the input places according to the data they hold, and set appropriate data in the generated tokens. We also make use of the fact that CPNTools supports time by attaching timestamps to tokens, which can be incremented by the transitions.

We have used PaMoMo to express different requirements for this transformation. Fig. 38 shows some of the specified invariants. The one to the left expresses how parallel servers should be translated into CPNs. In particular, if the parallelism of the server is `P`, then we need to replicate `P` times the CPN structure inside the set `servers` to the right. This is indicated by the expression `servers.size()=P`. The input and output blocks of the parallel server can be of any type, hence the invariant uses objects of type `Block` (represented by dotted rectangles) for them, to mean "any subclass of `Block`". Moreover, the labels `LS` and `LT` of these two blocks are used to locate the CPN places generated from them.

The upper right of Fig. 38 shows an invariant formalizing the translation of switches (both `TypeSwitch` and `SizeSwitch`). They should be transformed into places with as many output arcs as paths leaving from the switch. Finally, the bottom right of the figure shows an invariant describing the relation between the number of resources produced by a resource manager (with label `RM`) and the number of arcs that the corresponding transition should map to the place created for the resource. In particular, a correct transformation should produce as many arcs as the attribute `num` of the resource manager.

Altogether, in this complex case study we intensively used invariants with sets. This is due to the fact that both metamodels exhibited large heterogeneities. In particular, it was often the case that an attribute in the process-interaction language (like the parallelism in servers, or the resources produced by resource managers) had to be translated into a number of replicated struc-

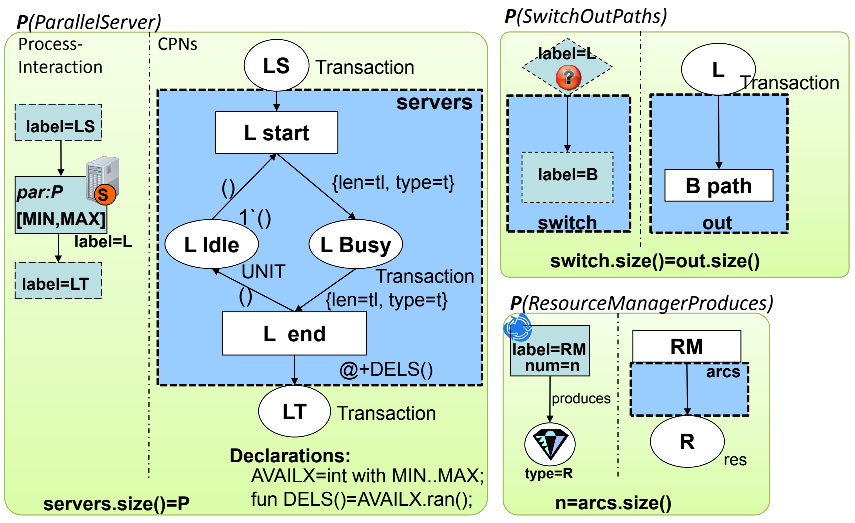


Fig. 38 Invariants for: translation of parallel servers (left), translation of switches (upper right), translation of number of resources produced by resource managers (bottom right)

tures in the CPN metamodel. Here we benefit from the fact that patterns are declarative, so that complex structures can be easily described graphically, as opposed to textually encoding them using e.g. OCL navigation expressions.

8.3 Verification of Graphical Definitions in the GMF

The Graphical Modeling Framework (GMF) [21] enables the "rapid" development of environments for visual languages. The approach taken is to specify different aspects of the editor using a set of interrelated models. The so-called `gmfgraph` model has a crucial role as it contains the specification of the graphical syntax of the language. However, only a tree-based editor is available for the specification of the figures of the concrete syntax, which is cumbersome and error prone. The `gmfgraph` model is then used (together with the other models) in a transformation to generate the so-called `gmfgen` model which is the basis for the final Java code generation of the editor.

A well-known problem is that if some model does not conform to a set of rules, the code generation produces erroneous code which may override a previous successful compilation. Although the framework validates some simple preconditions before the translation, like if all fields have meaningful values, no behavioral semantic contract is checked. Therefore, designers of GMF editors could greatly benefit from a means to check whether their models conform to the set of GMF norms that ensure a successful compilation. Here we use contracts for this purpose and discuss some preconditions.

Layout constraints. The specifications of figures need to provide a certain `Layout`, e.g., a `GridLayout` providing a row/column oriented layout. Typically, figures consist not only of a single figure but also contain children-figures, like labels to visualize feature values. The actual visualization of the children figures can be constrained by means of `LayoutData`. However, the type of `Layout` for a figure, e.g., `GridLayout`, should correspond to the type of `LayoutData` for its children, e.g., `GridLayoutData`. To check this, we can use the precondition in Fig. 39. The enabling condition selects figures with a certain `Layout` that contain a child figure with some `LayoutData`. Then, the OCL condition in the precondition checks the compatibility of the `Layout` and the `LayoutData`.

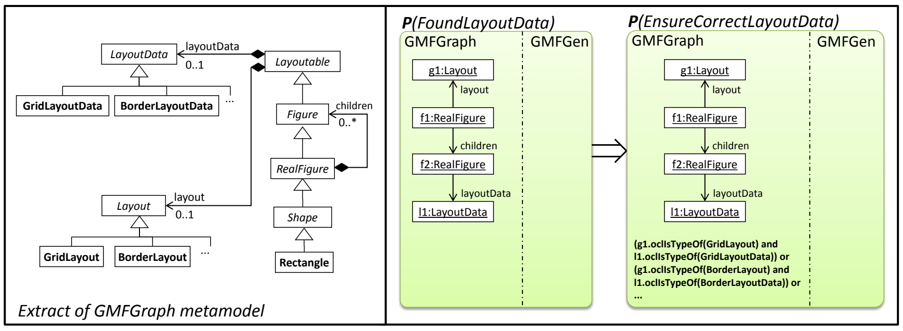


Fig. 39 Precondition checking layout constraints in GMF

Child access constraints. In order to be able to access the children figures of a figure in the `gmfmap` model, every `FigureDescriptor` (which describes a figure) needs to specify a `ChildAccess` for each one of its children. To be able to reuse a figure, e.g., if it is used several times in the concrete syntax, it is possible to use `Node`s assigning a graphical representation to a certain metamodel element. Hence, if a `Node` refers not only to a `FigureDescriptor` but also to some `ChildAccess`, then the figure referred by the `ChildAccess` must be a child of the `FigureDescriptor`. In addition, the type of the `Node` has to correspond to the type of the child figure (e.g., in case of a `DiagramLabel`, the type of the child figure must be `Label`). These two conditions can be checked by using the precondition shown in Fig. 40. The second condition is encoded by the OCL expression in the pattern.

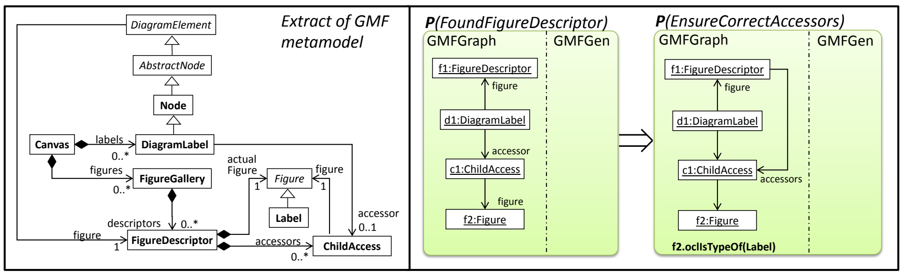


Fig. 40 Precondition checking child access constraints in GMF

Hence, this example shows the use of PaMoMo to make explicit certain (non-documented) assumptions of transformations. Once these assumptions are encoded in the form of preconditions, they can be checked using PaCo-Checker in order to avoid errors caused by the GMF compilation.

9 Related Work

The need for systematic verification of model transformations has been recognized by the research community and documented by several publications outlining the challenges to be tackled [3,4]. As a response, several verification approaches have been proposed, which may be classified into the following three areas: (i) verification of general properties such as confluence, applicability and termination of a set of transformation rules [10,33,51], (ii) automated generation of test input models [9,16,47], and (iii) verification of specific properties of transformations by means of oracle functions [41], which are used to analyse the validity of output models for a given set of test input models. The approaches dealing with the specification of oracles are the closest to the contributions of the present paper and are discussed in detail next.

In general, the literature distinguishes two kinds of oracle functions [4,41]. First, *complete oracle functions* may be defined by providing a full-fledged expected output model for each test input model, and subsequently, employing *model comparison* frameworks to verify the equality of the actual output model with respect to the expected model. Second, *partial oracle functions* expressed as *contracts* have been proposed for checking the validity of input models, output models, and their relationships. In addition, a third approach has recently been proposed [28], to specify oracle functions solely based on the *trace links* between the input models and output models. In the following, we elaborate on these three kinds, namely *verification by model comparison*, *verification by trace analysis*, and *verification by contract*.

Verification by model comparison. Complete oracle functions may be defined by having the expected output model at hand acting as a reference model for analysing the actual output model of a transformation as proposed in [31,36,37]. Model comparison frameworks are employed for computing a difference model between the expected and the actual output models. If there are differences then there is an error. However, reasoning about the cause for the mismatch solely based on the difference model (comprising differences such as additions, deletions, movements and updates of model elements) is challenging. Even more aggravating, several elements in the difference model may be caused by the same error, however, the transformation engineer has the burden to cluster the differences by himself. For large test input models expecting large output models, this approach seems unfeasible in practice, and partial oracle functions are more appropriate.

Verification by trace analysis. A complementary approach to model transformation testing has been proposed in [28] by using a *generic oracle function*. The idea of this approach is that the traces between the source and target models of a transformation should be similar to existing example traces. In particular, the oracle function checks how large a derivation of the generated traces of a model transformation from existing traces in the example base is. While this approach assumes that traces between source and target models exist, our approach aims at scenarios where no traces and even no corresponding target models for source models are available.

In [1], the authors use traces to trace back from a faulty output element the transformation rules causing the error. This idea can be incorporated into our framework, by the manual annotation of the transformation implementation rules with the invariants they are concerned with, as we did in [22].

Verification by contract. Contracts [39] are a well-established technique in software engineering to verify object-oriented programs [35]. Inspired by this work, contracts have also been applied for the verification of model transformations in previous research. In the following, we elaborate on several approaches proposed for verifying model transformations using contracts, divided into (i) OCL based, (ii) graph pattern based and (iii) model-fragment based approaches.

OCL based approaches. The first approach using contracts for model transformations was proposed by Cariou et al. [11,12]. The authors suggest implementing transformations with OCL. In this way, the source metamodel classes are provided with operations, which may comprise preconditions, postconditions and invariants. Although OCL natively supports design-by-contract, OCL is not intended to specify transformations and relationships between models. Thus, the authors propose an extension for OCL that allows defining mappings between input and output model elements.

A similar approach for defining contracts with OCL has been proposed in [40]. Besides other aspects, Kuester et al. [34] also agree on the use of OCL for the definition of transformation specific constraints for the produced output models. In [30], the authors propose the Epsilon Unit Testing Language to test model management operations. The language permits defining test operations where post-conditions for the model transformation under test may be specified. In a similar vein, Giner and Pelechano [19] propose a Test-Driven approach to the construction of model transformations. Thus, requirements for the transformation are captured in the form of test cases made of an input model together with output fragments and OCL assertions. Finally, in [20], a mechanism is presented to define properties for source models, target models, and source-target relationships as contracts expressed in OCL.

Graph pattern based approaches. In [2], the authors propose to use the patterns supported by the VIATRA2 tool to specify contracts for model transformations. However, their patterns operate on one model only, being therefore usable to specify pre- and postconditions, but not transformation invariants.

Model-fragment based approaches. A special form of verification by contract was presented in [41]. Based on [45], the authors propose to use model fragments for defining properties which are expected for an output model produced from a specific input model. For verifying these properties, the model fragments are matched on the produced output model. This approach is different from the previous ones, which propose using generic contracts solely defined on the metamodel level and not specific to a concrete test input model. The advantage of using model fragments is to support a user-friendly specification of test cases by reusing the graphical modeling editors, but this benefit comes with the price that the constraints are described at the model level. Thus, they have to be defined for each particular test input model.

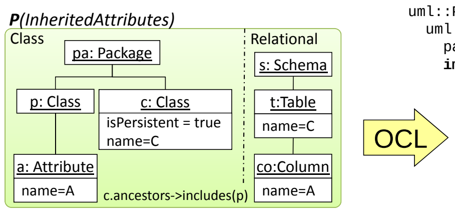
As in our proposal, all mentioned approaches (except the model-fragment based ones) define contracts based on the metamodel of the input and output models. However, the ones based on OCL usually lead to complex constraints, difficult to write in practice, and yielding verbose specifications [12], especially for the specification of relations between input and output models.

Finally, contracts have been used as oracle functions for testing object-oriented systems [8, 50]. In particular, these works aim at evaluating the *diagnosability* and the *robustness* or *vigilance* of systems provided with contracts. *Vigilance* refers to the degree in which contracts can detect faults in the running system. Diagnosability is related to the ease with which the faulty statements are found given a program failure. A relevant question is the level of detail required in contracts to find a significant number of failures and obtain high vigilance. Interestingly, both works agree that even contracts with low level of detail are good enough to find over 80% of software failures, being a good substitute for hand-crafted test oracle functions.

9.1 Contributions of PaMoMo and Discussion

Even though the community is spending considerable research effort on the verification and testing of transformations, and some approaches based on contracts have emerged, there is still the need for a high-level language able to express transformation properties. Therefore, in order to facilitate the specification of contracts, we proposed in this paper a visual language to define transformation contracts which induces several advantages.

First, our language is visual and enables a succinct expression of graph patterns, which otherwise would need to be encoded using navigation expressions in OCL, or complex expressions in the case of our notation for sets. This is illustrated in Fig. 41, which shows a positive invariant and the equivalent OCL expression for it (based on [22]). In particular, the OCL expression needs to include nested *forAll* clauses iterating over all instances of the source classes, and additional nested *exists* clauses checking the existence of appropriate objects in the target. Second, the semantics of our patterns is bidirectional, hence the same invariant may be used to verify a forward and a backward transformation. In contrast, using OCL one would need to encode differently the same invariant, iterating first the source elements with *forAll* clauses and then the target elements with *exists* clauses, or vice-versa. Although QVT-R (and other bidirectional languages) does not suffer from this drawback, we have seen that PaMoMo is more suitable (more succinct) than QVT-R to express contracts, most of all concerning the expression of negative information (a negative pat-



```
uml::Package.allInstances->forAll(pa |
  uml::Class.allInstances->forAll(p |
    pa.classes->includes(p)
    implies
      uml::Attribute.allInstances->forAll(a |
        p.attributes->includes(a)
        implies
          uml::Class.allInstances->forAll(c |
            (pa.classes->includes(c) and
             c.ancestors->includes(p) and
             c.isPersistent = true and p<>c)
             implies
               rel::Schema.allInstances->exists(s |
                 rel::Table.allInstances->exists(t |
                   s.tables->includes(t) and
                   rel::Column.allInstances->exists(co |
                     t.columns->includes(co) and
                     c.name = t.name and
                     a.name = co.name )))))))
```

Fig. 41 Positive invariant, and equivalent OCL code for its *forward* interpretation

tern produces two QVT relations), or enabling/disabling conditions (which generates additional QVT relations invoked from another relation).

Third, PaMoMo's formal semantics enables also reasoning about meta-model coverage, redundancies, contradictions and pattern satisfaction. Fourth, the specification of the contracts is completely decoupled from the transformation implementation. This means that the contracts are independent from the specified transformation rules and from the trace model of a specific transformation execution. Finally, the translation of the contracts to QVT-R allows for dedicated feedback in terms of the model elements not satisfying a particular contract. By using a pure OCL-based approach, only true or false is given back as answer to the user, but no further information is accessible in standard OCL environments (see Fig. 41). Hence, our approach provides better support for diagnosability than an approach based solely on OCL. In contrast to model-fragmentation, our approach allows the definition of the contracts with a visual language but we refrain from defining the contracts for a particular test input model.

Regarding the scalability of our approach, it depends on the size of the tested input and output models as well as on the size of the patterns, as we rely on a pattern matching mechanism. Thus, the smaller the models and patterns, the higher the performance. The size or complexity of the tested transformation implementation is not an issue though.

Some works report some limitations regarding the kind of failures that contracts can detect [50]. For example, in object oriented systems, method pre- and postconditions have difficulties in reasoning about the global state of an object or set of objects, as they are specified locally. For example, a *prune* method of a *stack* cannot define a trivial local contract checking if the removed element was previously inserted by a *put* method [50]. In contrast, transformation contracts are not specified at the rule level – as they are language-independent – and hence can be used to specify global transformation properties. In [50], it is argued that detecting certain failures requires overly complex contracts, more than the method implementation itself. In our case, contracts can be made more precise by: (a) enriching a pattern with enabling or disabling conditions, (b) adding more objects to the source or target compartments of a pattern, or (c) adding new patterns to the contract. It is up to future work to investigate the degree in which more complex contracts increase the effectiveness for failure detection (as in [8, 50]).

Finally, the kind of failures PaMoMo can detect is related to its expressiveness. There are some limitations concerning the specification of contextual conditions for a given property, as currently patterns support conjunction of disabling conditions but not disjunction or arbitrary boolean formulae over disabling conditions, or nested (i.e., recursive) conditions. The expresiveness of the graphical part of our patterns is limited (less than first-order logic). For example, we cannot model the absence of cycles of a given relation graphically. Nonetheless, in practice, we have found the expressiveness of PaMoMo to be enough to build useful contracts declaring interesting properties for our transformations, as we have shown in Section 8.

10 Conclusion and Future Work

Transformations should be developed using sound engineering principles. For this purpose, and based on the well-known design by contract paradigm, we have presented a visual, declarative language called PaMoMo to specify behavioral semantic contracts for M2M transformations in an implementation-independent way.

PaMoMo allows in a first step the specification of preconditions, i.e., conditions that need to be satisfied by input models to qualify for a transformation. Second, invariants are used to specify conditions that any pair of input/output models resulting from a correct transformation has to fulfill. Finally, postconditions are used to express required or forbidden configurations of elements in the output models. In order to make these declarative contracts operational, we reported on their compilation into check-only QVT-R transformations to check for conformance with respect to the specified contracts. Finally, a prototypical implementation was presented enabling the visual specification of contracts, their automatic compilation into QVT-R, and its chaining with the execution of the transformation under test. Several case studies were reported, showing the versatility of our approach.

In the future, we intend to work towards better facilities for error location (diagnosability) in the transformation implementation. In addition to annotating the implementation rules with the addressed invariants, a more complex but also more user friendly approach might be to employ heuristics exploiting trace information provided by the execution engines. This trace information might be matched with the error bindings provided by QVT-R to conclude on the rules causing the error. We also plan to study to which degree more detailed contracts (e.g., patterns with enabling or disabling conditions) improve failure detection in transformation testing. We are also investigating the use of the formal semantics of the contracts to analyse the compatibilities of individual transformations in a transformation chain. We also plan to extend the expressive power of PaMoMo, to use it for expressing contracts for in-place transformations, and to develop additional reasoning rules that help designers in building better contracts, e.g., by detecting mismatches between a contract and the metamodel constraints of the involved languages. It could be also interesting to integrate PAMOMO with our transformation engineering language *trans*ML [23] in order to apply the presented contract-based approach to industrial case studies. Moreover, our compilation into QVT-R opens the door to use PAMOMO not only as a specification language for contracts, but also as an executable, high level language to specify the actual transformation behavior. Finally, regarding tool support, we are working towards the use of the concrete syntax of models in PAMOMO specifications.

References

1. V. Aranega, J.-M. Mottu, A. Etien, and J.-L. Dekeyser. Using trace to situate errors in model transformations. In *Software and Data Technologies*, volume 50 of *Communications in Computer and Information Science*, pages 137–149. Springer, 2011.
2. A. Balogh, G. Bergmann, G. Csertán, L. Gönczy, Á. Horváth, I. Majzik, A. Pataricza, B. Polgár, I. Ráth, D. Varró, and G. Varró. Workflow-driven tool integration using model transformations. In *Graph Transformations and Model-Driven Engineering*, volume 5765 of *LNCS*, pages 224–248. Springer, 2010.
3. B. Baudry, T. Dinh-Trong, J. Mottu, D. Simmonds, R. France, S. Ghosh, F. Fleurey, and Y. Le Traon. Model transformation testing challenges. In *ECMDA Workshop on Integration of Model Driven Development and Model Driven Testing*, volume 92, 2006.
4. B. Baudry, S. Ghosh, F. Fleurey, R. France, Y. Le Traon, and J.-M. Mottu. Barriers to systematic model transformation testing. *Communication of the ACM*, 53:139–143, 2010.
5. A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins. Making components contract aware. *Computer*, 32:38–45, 1999.
6. J. Bézivin. On the unification power of models. *Software and Systems Modeling*, 4(2):31, 2005.
7. J. Bézivin, B. Rumpe, A. Schürr, and L. Tratt. Model Transformations in Practice Workshop of MoDELS'05, 2005.
8. L. C. Briand, Y. Labiche, and H. Sun. Investigating the use of analysis contracts to improve the testability of object-oriented code. *Softw., Pract. Exper.*, 33(7):637–672, 2003.
9. E. Brottier, F. Fleurey, J. Steel, B. Baudry, and Y. L. Traon. Metamodel-based test generation for model transformations: an algorithm and a tool. In *ISSRE'06*, pages 85–94. IEEE CS, 2006.
10. J. Cabot, R. Clarisó, E. Guerra, and J. de Lara. Verification and validation of declarative model-to-model transformations through invariants. *Journal of Systems and Software*, 83(2):283–302, 2010.
11. E. Cariou, N. Belloir, F. Barbier, and N. Djemam. OCL contracts for the verification of model transformations. *ECEASST*, 24, 2009.
12. E. Cariou, R. Marvie, L. Seinturier, and L. Duchien. OCL for the specification of model transformation contracts. In *Workshop on OCL and Model Driven Engineering @ UML'04*, volume 12, pages 69–83, 2004.
13. K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645, 2006.
14. EMF. Eclipse Modeling Framework. `www.eclipse.org/emf`. Last accessed: July 2011.
15. G. S. Fishman. *Discrete-Event Simulation: Modeling, Programming, and Analysis.* Springer, 2001.
16. F. Fleurey, B. Baudry, P.-A. Muller, and Y. Traon. Qualifying input test data for model transformations. *Software and Systems Modeling*, 8:185–203, 2009.

17. F. Fleurey, J. Steel, and B. Baudry. Validation in model-driven engineering: testing model transformations. In *MoDeVa'04*, pages 29–40. IEEE CS, 2004.
18. R. France and B. Rumpe. Model-driven development of complex software: A research roadmap. In *FOSE'07*, pages 37–54. IEEE CS, 2007.
19. P. Giner and V. Pelechano. Test-driven development of model transformations. In *MODELS'09*, volume 5795 of *LNCS*, pages 748–752. Springer, 2009.
20. M. Gogolla and A. Vallecillo. *Tract*able model transformation testing. In *ECMFA'11*, volume 6698 of *LNCS*, pages 221–235. Springer, 2011.
21. R. C. Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Addison-Wesley Professional, 2009. See also `http://www.eclipse.org/modeling/gmp/`.
22. E. Guerra, J. de Lara, D. S. Kolovos, and R. F. Paige. A visual specification language for model-to-model transformations. In *VL/HCC*, pages 119–126. IEEE CS, 2010.
23. E. Guerra, J. de Lara, D. S. Kolovos, R. F. Paige, and O. M. dos Santos. Engineering model transformations with *trans*ML. *Software and Systems Modeling*, In press, 2011.
24. D. Jackson. *Software Abstractions. Logic, Language, and Analysis*. MIT Press, 2006.
25. K. Jensen. *Coloured Petri nets basic concepts, analysis methods and practical use (Monographs in theoretical computer science)*. Springer, 1997.
26. K. Jensen, L. M. Kristensen, and L. Wells. Coloured Petri nets and CPN tools for modelling and validation of concurrent systems. *STTT*, 9(3-4):213–254, 2007.
27. F. Jouault and I. Kurtev. Transforming models with ATL. In *Model Transformations in Practice Workshop*, 2005.
28. M. Kessentini, H. A. Sahraoui, and M. Boukadoum. Example-based model-transformation testing. *Autom. Softw. Eng.*, 18(2):199–224, 2011.
29. D. Kolovos, R. Paige, and F. Polack. The Epsilon Transformation Language. In *ICMT'08*, volume 5063 of *LNCS*, pages 46–60. Springer, 2008.
30. D. Kolovos, R. Paige, L. Rose, and F. Polack. Unit testing model management operations. In *ICSTW'08*, pages 97–104. IEEE CS, 2008.
31. D. S. Kolovos, R. F. Paige, and F. A. Polack. Model comparison: a foundation for model composition and model transformation testing. In *GaMMa'06*, pages 13–20. ACM, 2006.
32. T. Kühne. Matters of (meta-)modeling. *Software and Systems Modeling*, 5(4):369–385, 2006.
33. J. M. Küster. Definition and validation of model transformations. *Software and Systems Modeling*, 5(3):233–259, 2006.
34. J. M. Küster and M. Abd-El-Razik. Validation of model transformations - first experiences using a white box approach. In *Models in Software Engineering*, volume 4364 of *LNCS*, pages 193–204. Springer, 2006.
35. G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. *Sci. Comput. Program.*, 55(1-3):185–208, 2005.
36. Y. Lin, J. Zhang, and J. Gray. Model comparison: A key challenge for transformation testing and version control in model driven software development. In *OOPSLA Workshop on Best Practices for Model-Driven Software Development*, 2004.
37. Y. Lin, J. Zhang, and J. Gray. A testing framework for model transformations. *Model-Driven Software Development*, pages 219–236, 2005.
38. T. Mens and P. Van Gorp. A taxonomy of model transformation. *ENTCS*, 152:125–142, 2006.
39. B. Meyer. Applying "design by contract". *Computer*, 25:40–51, 1992.
40. J.-M. Mottu, B. Baudry, and Y. Le Traon. Reusable MDA components: A testing-for-trust approach. In *MoDELS'06*, volume 4199 of *LNCS*, pages 589–603. Springer, 2006.
41. J.-M. Mottu, B. Baudry, and Y. L. Traon. Model transformation testing: oracle issue. In *ICSTW'08*, pages 105–112. IEEE CS, 2008.
42. Object Management Group. OCL Specification Version 2.0. `http://www.omg.org/docs/ptc/05-06-06.pdf`, 2005.
43. Object Management Group. QVT Specification Version 1.1. `http://www.omg.org/spec/QVT/1.1/`, 2011.
44. OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. `http://www.omg.org/spec/QVT/1.1/Beta2/PDF/`, 2009.

45. R. Ramos, O. Barais, and J.-M. Jézéquel. Matching model-snippets. In *MoDELS'07*, volume 4735 of *LNCS*, pages 121–135. Springer, 2007.
46. D. C. Schmidt. Model-Driven Engineering. *IEEE Computer*, 39(2):25–31, 2006.
47. S. Sen, B. Baudry, and J.-M. Mottu. Automatic model generation strategies for model transformation testing. In *ICMT'09*, volume 5563 of *LNCS*, pages 148–164. Springer, 2009.
48. J. M. Spivey. An introduction to Z and formal specifications. *Softw. Eng. J.*, 4(1):40–50, 1989.
49. TATA Research Development and Design. ModelMorf. `http://www.tcs-trddc.com/trddc_website/ModelMorf/ModelMorf.htm`. Last accessed: July 2011.
50. Y. L. Traon, B. Baudry, and J.-M. Jézéquel. Design by contract to improve software vigilance. *IEEE Trans. Software Eng.*, 32(8):571–586, 2006.
51. D. Varró, S. Varró-Gyapay, H. Ehrig, U. Prange, and G. Taentzer. Termination analysis of model transformations by Petri nets. In *ICGT'06*, volume 4178 of *LNCS*, pages 260–274. Springer, 2006.
52. Xpand. Xpand templates:. `http://wiki.eclipse.org/Xpand`. Last accessed: July 2011.