Software and Systems Modeling manuscript No.
(will be inserted by the editor)

Inter-Modelling with Patterns

Esther Guerra[1] ⋆, **Juan de Lara**[1], **Fernando Orejas**[2]

[1] Universidad Autónoma de Madrid (Spain), e-mail: {Esther.Guerra, Juan.deLara}@uam.es
[2] Universitat Politècnica de Catalunya (Spain), e-mail: orejas@lsi.upc.edu

Received: date / Revised version: date

Abstract *Inter-modelling* is the activity of modelling relations between two or more models. The result of this activity is a model that describes the way in which different models can be related. Many tasks in Model Driven Development can be classified as *inter-modelling*, for example designing model-to-model transformations, defining model matching and traceability relations, specifying model merging and model weaving, as well as describing mechanisms for inter-model consistency management and model synchronization.

This paper presents our approach to inter-modelling in a declarative, relational, visual, and formal style. The approach relies on declarative patterns describing allowed or forbidden relations between two models. Such specification is then compiled into different operational mechanisms that are tailor-made for concrete inter-modelling scenarios. Up to now, we have used the approach to generate forward and backward transformations from a pattern specification. In this paper we demonstrate that *the same* specification can be used to derive mechanisms for other inter-modelling tasks, such as model matching and model traceability. In these scenarios the goals are generating the traces between two existing models, checking whether two models are correctly traced, and modifying the traces between two models if they are incorrect.

Key words Inter-Modelling – Model-to-Model Transformation – Model Matching – Traceability – Graph Transformation – Graph Constraints

1 Introduction

Models are the heart and soul of Model Driven Engineering (MDE). They are used to specify, analyse, design, test, document, maintain and generate code for the applications to be built. Frequently, engineers use several modelling languages and models during the development process, which cannot remain oblivious of each other and have to co-evolve together. Hence, many common management activities in MDE involve manipulating several models at a time, like model-to-model transformation (batch or incremental) [11,25,38,46], inter-model consistency [22] and synchronization [21], model traceability [14], model matching [29,30] and model merging [5, 7]. Unfortunately, the specifications used in each of these activities are often built separately — even if they involve the same kind of models — and use different languages and tools. This results in an increase of development time and effort, and may lead to inconsistencies between the specifications.

We define *inter-modelling* as the activity of modelling relations between two or more models. The result of this activity is a model that describes the way in which different models can be related. The term emphasizes the fact that these relations are described by means of *models*, which as such are amenable to analysis. Many MDE tasks can be interpreted in terms of *inter-model models* (or *inter-models* in short). For example, in a simple model-to-model transformation scenario, a source model is transformed into a target model. Equivalently, this scenario can be formulated as finding a target model such that, together with the source, conforms to a specific inter-model and the relations it defines. Further use cases of inter-models include specifications for model matching, model merging, inter-model consistency and model synchronization.

In previous works [11,25,38] we proposed a declarative, relational and formal approach to Model-to-Model (M2M) transformation based on *triple patterns* that express the relations between two models. Our patterns are similar to graph constraints [16] but for triple graphs made of two graphs plus their traceability relations. Patterns can specify positive information (the relation they declare must hold) or negative information (the relation

Send offprint requests to:
⋆ *Present address:* Universidad Autónoma de Madrid, Campus Cantoblanco, 28049 Madrid, Spain

must not hold). In [25], we compiled such pattern specifications into operational mechanisms implemented with Triple Graph Grammar (TGG) operational rules [15,33, 43] to perform forward and backward transformations.

In this paper, we propose our patterns as a unifying, formal framework for inter-modelling by their compilation into different operational mechanisms that are tailor-made for solving concrete MDE scenarios. In particular, in addition to M2M transformation, we demonstrate the use of the specifications for the problems of model matching and model traceability. For this purpose, our patterns are used as a high-level specification to check whether two models are correctly traced, and generate low-level mechanisms to delete incorrect traces and create those that are missing. The advantage of this framework is that *the same* specification can be used for M2M transformation, model matching and model traceability. Moreover, its formal basis enables analysis at the pattern and operational levels.

The paper also details how we handle attributes, as this has been one of the main difficulties of declarative bidirectional languages. Since attribute computations must be specified in a non-causal way in this kind of languages, the generation of operational mechanisms usually involves the algebraic manipulation of these non-causal expressions to synthesize attribute pre-conditions and computations, which may be difficult to automate. We tackle this issue by the uniform integration of attribute computations and conditions in patterns, and by considering the manipulated models also as constraints, hence avoiding algebraic manipulation. Thus, during the transformation, attributes in models are specified by variables and formulae constraining them. When the transformation ends, one can resort to an equation solver to obtain concrete attribute values.

Our proposal has the following benefits. First, it provides a relational language for inter-modelling which unifies different inter-modelling scenarios in a single specification. This solves the problem of maintaining different, heterogeneous, scattered specifications for different scenarios that involve the same kind of models. Separating the operational mechanism from the declarative specification allows generating specific operational mechanisms for different purposes (transformation, update, matching, etc.) as well as using different operational languages (graph grammar rules [25], Coloured Petri nets [12], constraint solvers [8] or QVT-Core [40]). Second, its relational style contrasts with declarative approaches such as TGGs, which are designed to be used with grammar-based languages, and where a causality between the existing elements in the models and the ones to be created has to be given. Our language is suitable for meta-model-based languages, and the interpretation of patterns as constraints makes it possible to check the consistency of models against the specification. A third advantage of our proposal is that the order of pattern enforcement is deduced, contrary to approaches such as QVT-Relations, where it must be explicitly specified. Fourth, its formal foundation allows studying the specification in both declarative (patterns) and operational (derived rules) formats. This contrasts with most of the current approaches [3,30,32,40], where the high-level languages are only semi-formally defined. In our view, the MDE community would benefit from a clean, formal semantics for their transformation languages enabling the analysis of transformations and serving as a reference for tool builders. Finally, our patterns have a compositional style, i.e. models satisfy two patterns in conjunction if they satisfy the two patterns separately. This makes specifications extensible.

As a side effect of our compilation into graph transformation rules, we have introduced a new approach to graph transformation where attribute computations are expressed as constraints. This has value in itself, as it allows for the loose specification of transformations and the exploration of valid solutions.

This paper extends the conference paper [25] as follows. It presents patterns as a language for inter-modelling, and it considers other usage scenarios for them, namely model matching and model traceability. We extend patterns with constraints that can contain arbitrary formulae and abstract objects, in order to write more compact specifications. Finally, we consider the interplay between patterns and meta-models, as the latter induce extra constraints.

Paper organization. Section 2 overviews current inter-modelling approaches, pointing out limitations and needs in the area. Section 3 explains the scheme of our proposal. Section 4 introduces the basic algebraic notions to be used in the remaining sections. Then, Section 5 presents our patterns for inter-modelling. Section 6 shows the compilation of patterns into TGG rules for M2M transformation, taking into account meta-model integrity constraints, and sketching some heuristics to improve their efficacy. Next, Section 7 shows the use of patterns for model matching and model traceability, presenting two notions of satisfaction, the generation of relating rules, and the procedures for detecting and removing incorrect traces. Section 8 presents a complete case study, and Section 9 ends with the conclusions. An appendix gives details of the main claims and propositions of the paper.

2 Related Inter-Modelling Approaches

We define *inter-modelling* as the activity of modelling relations between models. Hence, we believe it is important to produce models describing such relations, as opposed to hard-coding specific mechanisms for each concrete scenario. The use of models has the advantage of being more flexible, understandable and maintainable than lower-level programs. Inter-models can be seen as a generalization of the term *transformation-model* [4], but

comprise additional inter-modelling scenarios other than transformations.

As previously stated, inter-modelling spawns a large variety of situations, like specifying M2M transformations, model merging, model weaving, model composition, model matching, model tracing, inter-model consistency and model synchronization. In this section we do not aim at being exhaustive, but we focus on M2M transformation, model matching and model traceability, as these are the scenarios to which we will compile our patterns in this paper. In the following, we introduce current approaches and some of their advantages and drawbacks. The observed needs will lead to the challenge of a unifying inter-modelling framework.

Model-to-model transformation. M2M transformation is one of the most common activities in MDE. It is used to migrate between language versions, to transform into verification domains, to refine models (for implementation) or to abstract models (for re-engineering).

There are two main approaches to M2M transformation: *operational* and *declarative*. The former is based on operations that explicitly state how and when to create target elements from source elements [3,32,40,42, 45]. They often borrow imperative constructs from traditional programming languages, such as statements for iteration. This makes them low-level and difficult to analyse, but at the same time they provide great flexibility, even at the cost of verbosity. Most approaches [3,32,42] provide rule-like built-in constructs that iterate on all elements of a certain type on the source domain, leaving the definition of application conditions to the user. Moreover, different usage scenarios imply developing different transformations, whether or not they involve the same modelling languages and relations.

On the contrary, declarative approaches usually describe mappings between source and target models in a direction-neutral way, which can be naturally interpreted as a specification of the relations that must be satisfied by two models to be considered consistent. They often allow declaring patterns in the relations, instead of single elements. From this high-level specification, operational mechanisms are generated for different transformation scenarios, e.g. to transform a source model into a target one or vice versa (forward and backward transformations), for incremental transformations [21], or to synchronize two models [33]. Our patterns for inter-modelling belong to this category.

Declarative languages can be categorized with attention to certain features. First, some languages can explicitly create and query traces, while others cannot. The first kind of languages includes TGGs [43], Tefkat [34] and our patterns [11], whereas QVT-Relations [40] falls in the second category. Being able to query the traces and check which elements are related across models can be used as a pre-condition for other relations. This feature may be seen as low-level, but it is essential in model traceability scenarios. Those languages that do not permit an explicit handling of traces use parameter passing and explicit relation invocation instead. Another interesting feature is whether the specifications in a given language impose a direction, or whether they are direction-independent and can be interpreted both ways without changing the specification. An additional feature that a language can have or not is the possibility of expressing non-constructive constraints, e.g. forbidding the existence of certain structures or asserting that a certain property does not have a particular value. We can also look at the uses of the specifications. Here we can differentiate between specifications that can only be used to produce low-level mechanisms for specific activities (like forward or backward transformation), and specifications that in addition can be used in a higher level way to test whether two models are consistent (what we call *checkonly* scenarios). Finally, if the semantics of the language is formally defined, specifications are amenable to analysis.

Among the declarative approaches, a prominent example is QVT-Relations (QVT-R) [40], the highest-level of abstraction language of the QVT OMG standard [40]. In this language, a M2M transformation is made of relations with two or more domains. These are described by patterns similar to object diagrams. When a domain is marked as *enforced*, the models to which it is applied may be modified in order to satisfy the relation; whereas if it is marked as *checkonly*, they are just inspected to check for disagreements. Unfortunately, such marking is done at the specification level, which mixes specifications with their intended operational usage. In any case, the standard prescribes that such operational scenarios are to be performed by the compilation of the QVT-R transformation into QVT-Core. If the transformation is executed in the direction of a check-only domain, the model is not modified, but it is checked wether it is consistent with the source model according to the transformation specification. In QVT-R, traces are not explicitly handled, but relations may contain *when* and *where* clauses instead. The former express conditions under which the relation needs to hold, and usually refer to other relations to which they pass parameters. *Where* clauses may call other relations, similar to function calls in traditional programming. QVT-R does not permit non-constructive constraints as it does not define the notion of negative relation. Although QVT-R could be used for additional M2M transformation scenarios, the standard does not describe that possibility. Finally, QVT-R lacks a formal semantics, which hinders analysis of transformation properties.

Tefkat [34] is another declarative M2M transformation language, based on logics. Its specifications are made of parameterized rules that consist of two constraints – source and target – sharing variables. Each rule matches and constrains a number of objects, either from the source model or from the trace, and then cre-

ates (or ensures the existence of) a number of target elements with a set of constraints. Rules can both invoke other rules with parameters as well as handle explicitly the trace. Tefkat specifications are directed, as they can only create elements in the target model. They can only be used operationally, not allowing the test for consistency, and do not admit negative constraints. Tefkat has a formal semantics based on logics, although no verification support is provided by the authors.

In [1], the authors describe declarative transformations using the mathematical notion of relation, which is used to explicitly represent the trace model. Generic relations are encoded in a meta-model with OCL constraints, which serves as the basis for building inter-models. Such inter-models express the relationships between the elements in two models using OCL invariants, which must be provided by the engineer. Mapping correctness is evaluated using the Kent Modelling Framework [28], but reconciliation of source/target models has to be encoded manually. Specifications are direction independent, but do not support negative patterns. This approach is similar to the notion of *transformation model* [4], and hence our patterns could serve as a higher-level model from which such transformation models could be derived, along the lines of [8].

BOTL [6] is a formal M2M transformation language based on rules, each having source and target object patterns with variables. Rules do not have an explicit notion of trace, nor is there parameter passing between rules. Instead, they are applied independently, and the objects created with same identifier are merged. Transformations in BOTL are bidirectional. The considered operational scenarios are just forward and backward transformations, and there is no support for negative constraints.

TGGs [43] formalize the synchronized evolution of two graphs through declarative rules from which TGG operational rules are derived. These operational rules solve different scenarios, like forward and backward transformations or model matching. The approach is suitable for languages defined through creation graph grammars but not through meta-modelling. This is so because the generated operational rules cannot be applied as long as possible, but need a control mechanism to guide their execution. Frequently, the order in which the creation rules were applied to create the source model is used to apply the operational rules [15], which may imply having to parse the source model before applying the transformation. Ad-hoc solutions like rule priorities were also proposed in [33] as a control mechanism. TGGs are based on graph transformation, thus they are formally defined and allow verification [15]. They are direction independent but, although they have been recently extended with Negative Application Conditions [17,18,44], it is not possible to express forbidden global conditions. Finally, being based on rules, specifications can be used for the checkonly scenario, but one has to resort to parsing.

Several declarative languages have been defined for bidirectional updating, many of them based on *lenses* [20]. These are well-behaved bidirectional transformations that operate on ad-hoc, textual data formats. For example, in [36] the authors start from a forward transformation and the corresponding backward transformation is derived. Their transformations work on trees (e.g. derived from parsing XML documents) and only contain injective functions to ensure bidirectionality. If an attribute can take several values, one of them is chosen randomly. Other works borrow concepts from formal languages and compilers. For example, in [13] attribute grammars are used as a transformation language, where the order of execution of rules is derived from attribute dependencies. Finally, other works, like [41] tackle the deficiencies of attribute computations in standard graph transformations, but fail to be useful for bidirectional transformations.

In short, after this overview of M2M transformation languages, we observe a wealth of approaches, languages and tools, but a lack of formally defined languages, usable on meta-modelled languages, allowing for direction independent specifications, permitting specialized compilations into different languages, and solving different scenarios. In this paper we propose one such language.

Model matching. Model matching or comparison is the activity of comparing two models and matching the elements considered similar, according to some specification [29]. The generated traces can be used in a variety of ways, for example to merge the models or to check if they are synchronized. The matching specification can be seen as a model that describes inter-model consistency conditions. Then, the trace-generating operational mechanisms create traces between the model elements so that they conform to the matching specification. Some approaches exist for comparing two models expressed in the same language, typically UML [37,48]. However, the customization of how comparisons are made is usually limited and, in addition, the advent of Domain-Specific Languages has made evident the need for comparing heterogeneous models. Although some rule-based, dedicated languages have been proposed for comparison [30], specifications in these languages have to be kept consistent with other specifications, perhaps for M2M transformation between the same languages. This produces redundancy and introduces the possibility of incongruities between the different specifications. Moreover, no approach to model matching provides a formal foundation enabling the analysis of specifications. Even though TGGs can be used for model matching [33], the compilation into operational mechanisms does not produce application conditions, and hence extra control mechanisms may have to be designed in an ad-hoc manner for this particular scenario.

Model traceability. Similar to model matching, model traceability [9,14,19,35] articulates the dependencies between the different models created during software development. Sometimes such trace information is automatically generated, e.g. if one of the models is generated from the others via a M2M transformation. Other times, traceability links have to be established manually, e.g. to trace requirements into further design models [2]. Traceability information can be seen as a model in its own right [14], and current research is revolving around semi-automatic methods to establish and maintain such traceability relations [27,35]. Even though TGGs could be used for creating traceability links, one still needs to design a specific control mechanism to guide the execution of the rules.

Conclusion. After this overview, we notice that many inter-modelling languages exist, but few are able to solve more than one inter-modelling scenario. Thus, a challenge in this area is to provide a unifying framework in which a unique inter-model specification can be used to solve several operational scenarios such as M2M transformation, model matching, and model synchronization. Moreover, defining such a framework formally would enable the analysis of the specifications, and perhaps of the operational mechanisms too, for each particular scenario. In the rest of the paper we provide a detailed account of such a framework.

3 Our Inter-Modelling Architecture

Fig. 1 provides the general scheme of our approach. In step 1, the designer builds the inter-model specification using our pattern language. Our patterns have an underlying formal foundation which makes them amenable to analysis (step 2). For example, we can investigate: (i) pattern conflicts, as a pattern may forbid a relation which is required by other pattern; (ii) conflicts of patterns with respect to the language meta-models, e.g. if a pattern requires two links stemming from an object but the meta-model cardinality constraints only allow one; and (iii) meta-model coverage, by inspecting the elements that patterns use and create, as well as the unused types. In this paper we will not go deeper in these analysis techniques, but on the fundamentals of our approach.

In step 3, the designer chooses the usage scenario for the specification: transformation, model matching or model traceability. For each one of these scenarios we have developed: (i) a formal notion of satisfaction which detects whether two related models satisfy the specification for the particular scenario; and (ii) operational mechanisms, based on graph transformation rules, which manipulate the input model(s) and their traces in order to make them satisfy the specification. In the transformation scenario, the designer can decide whether the operational mechanisms are for forward or backward transformation, as our patterns are direction-independent and can be interpreted both ways. In this case the synthesized mechanisms will create a target model from a source one from scratch (forwards) or vice-versa (backwards). For model matching and model traceability, the generated operational mechanisms are able to create appropriate traces between the compared models, as well as to delete incorrect traces.

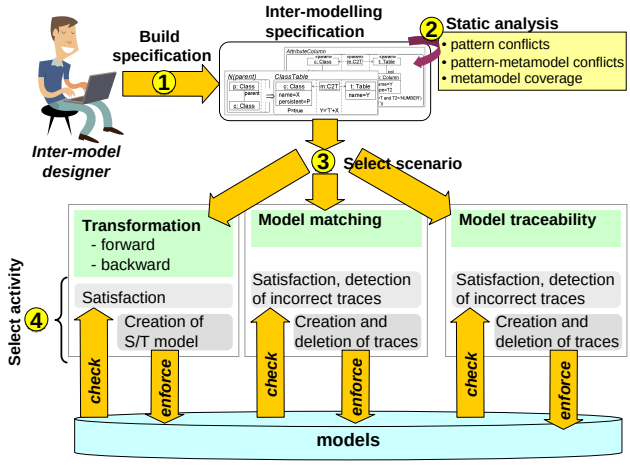

Fig. 1 Scheme of our inter-modelling approach.

By construction our operational mechanisms are: (i) complete (able to generate all possible target models that together with the source satisfy the specification), (ii) correct (the result satisfies the specification), and (iii) terminating (the execution as long as possible of the generated rules is finite no matter the specification and initial model) [38].

Before explaining the details of our pattern language, next section introduces some necessary concepts.

4 Preliminaries

This section introduces some technical preliminaries: triple graphs, constraint triple graphs, and triple graph transformation.

Triple graphs are the structures we use to store the two models we want to relate, as well as a third model which contains traces (similar to pointers) relating one element in each one of the other two models. Formally, this triple structure is made of three graphs, and two mappings (technically, graph morphisms[1]) from the trace graph to the other two graphs. Note that graphs can be provided with a rich structure allowing for a precise mathematical representation of models. In particular, we use labelled graphs (called E-graphs in [16]), which are graphs with data in nodes and edges. Mappings between E-graphs (morphisms) are tuples of set

[1] A morphism corresponds to the mathematical notion of function, but instead of plain sets, we consider graphs.

mappings – one for each set in the E-graph – such that the structure of the E-graph is preserved [16]. For the typing we use a type graph TG [16], similar to a meta-model. In this way, typed graphs become tuples $(G, type\colon G \to TG)$, where *type* is a typing morphism from G to TG.

Altogether, triple graphs are made of three graphs: *source* (S), *target* (T) and *correspondence* (C). Nodes in the correspondence graph relate nodes in the source and target graphs by means of two graph morphisms [15]. For technical reasons, we restrict nodes in the correspondence graph to be unattributed. This is a side effect of the current formulation, whose aim is simplicity, but we could use a richer trace model as we did in [23]. The correspondence graph, however, is a graph in its own right, and hence it may include edges, too.

We use the notation $\langle S, C, T\rangle$ for a triple graph made of graphs S, C and T, leaving the mappings between S, C, and T implicit. Given a triple graph $TrG = \langle S, C, T\rangle$, we write $TrG|_X$ for $X \in \{\{S\}, \{T\}, \{S, T\}\}$ to refer to a triple graph where only the graphs in X are present, i.e. $TrG|_S = \langle S, \emptyset, \emptyset\rangle$, $TrG|_T = \langle \emptyset, \emptyset, T\rangle$, and $TrG|_{ST} = \langle S, \emptyset, T\rangle$ (where $\emptyset$ is the empty graph).

Example. The lower part of Fig. 2 shows a triple graph relating a class diagram and a relational schema. The graph nodes $V^S = \{Class1, Class2, Attr1\}$, $V^C = \{C2T1, C2T2\}$ and $V^T = \{Table1\}$ are depicted as rectangles, and the data nodes denoting the values of the attributes as circles. The edges from the graph nodes to the data nodes assign values to the attributes of the graph nodes. For instance, $Table1$ has one attribute *name* with value '$c1$'. The upper part of the figure shows the meta-model triple used for typing. We have explicitly depicted the typing morphism with dashed arrows, but throughout the paper we will use the UML notation of object diagrams.

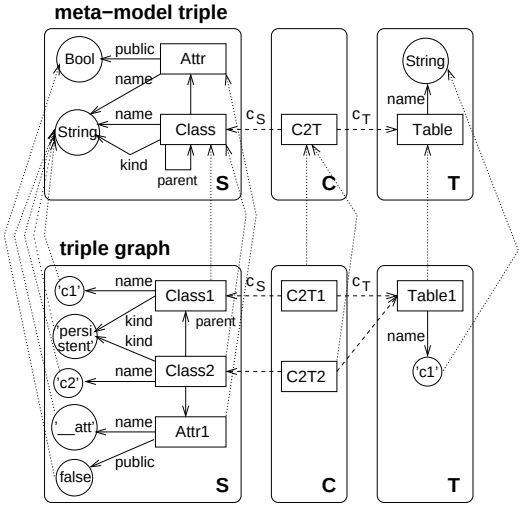

Fig. 2 Triple graph and meta-model triple examples.

In order to describe the manipulation of triple graphs by means of graph transformation rules, these rules may need to include graphs storing variables that will typically be instantiated when applying the rule. Moreover, we may need to express some properties about these variables. We have formalized this kind of graph using the new notion of *constraint triple graph*. This is a triple graph attributed over a finite set of variables, and equipped with a formula on this set to constrain the possible attribute values of source and target elements. We use the notation $\langle TrG, \alpha\rangle$, where TrG is a triple graph whose labels are variables and α is a formula over these labels, to denote a constraint triple graph $CTrG$. Moreover, given $CTrG$ we denote by α_{CTrG} its associated formula.

Example. Fig. 3 shows a constraint triple graph. We take the convention of placing in the left compartment the terms of the formula concerning only source graph attributes; in the right compartment the terms constraining only attributes in the target; and the terms constraining both in the middle. In all cases we omit the conjunctions. Hence, in the example, the formula α is $y = x \times 2 \wedge y > 0 \wedge z = 3$. Note that "=" denotes equality, not assignment. Hence, in our approach there is no attribute computation, but only attribute conditions. Finally, we omit unused attributes in the figures.

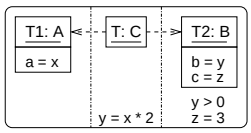

Fig. 3 Constraint triple graph example.

As an extension to [25], we allow for arbitrary first-order formulae in constraints. For example, the previous constraint could contain the following formula (assuming all variables of sort $\mathbb{N}$): $\forall v \nexists w[(v > 1 \wedge v < x \wedge w > 1 \wedge x/v = w) \Rightarrow (y = x \times 2 \wedge z = 3)] \vee (y = x \times 3 \wedge z = 3)$, meaning that z should be 3, and y the triple of x; moreover if x is prime then y is allowed to be its double. In practice, we use OCL expressions in constraints.

Notice that whereas triple graphs store data in nodes as labels (see Fig. 2), this is not so in constraint triple graphs, where the labels are variables. Thus, if we want to store a value V on a node of a constraint triple graph, it is enough to label that node with some fresh variable x and include the equality $x = V$ in the associated formula. This is illustrated in Fig. 3, where the term $z = 3$ makes 3 the unique possible value for $T2$'s attribute c.

Sometimes, we may need to restrict formulae over a set of variables ν to a smaller set of variables $\nu' \subseteq \nu$, for example when restricting a constraint triple graph to the source or target graph only. Thus, given a formula α over ν, its restriction to $\nu' \subseteq \nu$ is given by $\alpha|_{\nu'} = \exists(\nu \setminus \nu')\alpha$, where we have existentially quantified all variables in ν which are not in ν'. This is done because if one thinks

of a formula as a set of constraints on variables, then adding an existential quantifier on one variable relaxes the constraints on that variable so that they are "ignored". In our case we relax the variables in $\nu \setminus \nu'$. For example $(x = 5 \wedge y = x+1)|_{\{x\}} = \exists y[x = 5 \wedge y = x+1]$, and $(x = 5 \wedge y = x+1)|_{\{y\}} = \exists x[x = 5 \wedge y = x+1]$. The first restriction obviates y, while the second obviates x.

Given a constraint triple graph $CTrG$ consisting of the triple graph TrG and the formula α, we write α^S for the formula α restricted to the source variables, and α^T for the restriction to the target variables.

In order to relate constraint triple graphs we use a notion of embedding (technically, a constraint triple graph morphism). In particular, $CTrG = \langle TrG, \alpha \rangle$ is embedded in $CTrG' = \langle TrG', \alpha' \rangle$, denoted $CTrG \hookrightarrow CTrG'$, if TrG is a subgraph of TrG' (up to variable renaming), α' implies α, and also the source and target restrictions of α' imply the source and target restrictions of α, respectively. Since $CTrG$ may be embedded in $CTrG'$ in several different ways (e.g. TrG may be a subgraph of TrG' at several different locations) we will distinguish the different embeddings between $CTrG$ and $CTrG'$ by assigning them a name and using a functional notation. This means that $e : CTrG \hookrightarrow CTrG'$ and $e' : CTrG \hookrightarrow CTrG'$ may be two different embeddings of $CTrG$ in $CTrG'$. In this sense, for an embedding $e : CTrG \hookrightarrow CTrG'$, sometimes we will say that $CTrG$ is embedded or included in $CTrG'$ at location e.

Example. Fig. 4 shows a constraint triple graph embedding (i.e. a morphism). The embedding renames variables x and y as x_0 and y_0, respectively. We may see that the triple graph on the left is a subgraph of the triple graph on the right, up to the variable renaming. Concerning the formula, we may see that the formula on the right constraint implies the formula on the left constraint, up to the given renaming, and that the source and target restrictions of the left formula imply the source and target restrictions of the right formula, respectively. For instance, for the target restriction we have that $y_0 >= 1$ implies $y_0 > 0$ (i.e. $y > 0$), and for the whole formula, that $(x_0 = 4) \wedge (z > x_0) \wedge (x_0 > y_0) \wedge (w > x_0) \wedge (y_0 \geq 1)$ implies $(x_0 > 0) \wedge (x_0 <> y_0) \wedge (y_0 > 0)$.

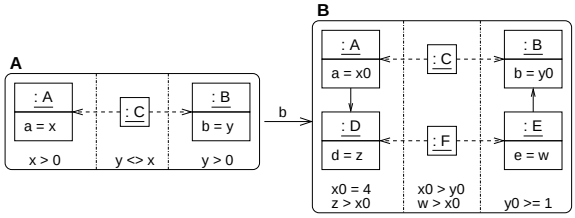

Fig. 4 Embedding of constraints.

As said above, sometimes we need to restrict a constraint triple graph to one of its components. For instance, we may want to consider only the source or the target graph of a constraint triple graph, or just to ignore its correspondence graph. This can be done as follows. Given a constraint triple graph $CTrG$ consisting of the triple graph TrG and its associated formula α:

- its source restriction is given by the triple graph $CTrG|_S$ consisting of the triple graph $TrG|_S$ and the formula α^S.
- its target restriction is given by $CTrG|_T$ consisting of the triple graph $TrG|_T$ and the formula α^T.
- its source-target restriction is given by $CTrG|_{ST}$ made of $TrG|_{ST}$ and the formula α.

Moreover, given an embedding $e : CTrG \hookrightarrow CTrG'$, we denote by $e|_S : CTrG|_S \hookrightarrow CTrG'|_S$, $e|_T : CTrG|_T \hookrightarrow CTrG'|_T$, and $e|_{ST} : CTrG|_{ST} \hookrightarrow CTrG'|_{ST}$, the corresponding restrictions of e.

As we will show later, we need to manipulate constraint triple graphs through *pushouts*, since this is the basis for graph transformation. A pushout is the result from gluing two objects (triple graph constraints in our case) B and C with respect to a common intersection A, written $B +_A C$. More precisely, a pushout can be seen as the union of B and C when A is its intersection. Pushouts between constraint triple graphs are built by making the pushout of the corresponding triple graphs, and taking the conjunction of their formulae.

Example. Fig. 5 shows the pushout of constraints B and C through their intersection A, to yield constraint D. In MDE terms, a pushout is similar to a model merging operation of two models (B and C) through some identified correspondences (model A).

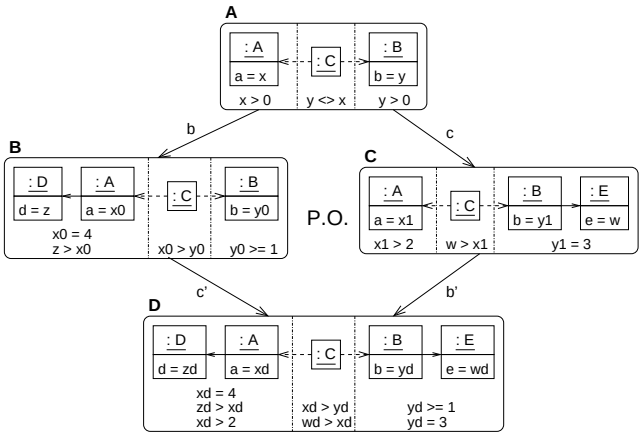

Fig. 5 Pushout example.

4.1 Transformation Rules

A useful observation is that an attributed triple graph can be seen as a constraint triple graph whose formula is satisfied by a unique variable assignment. We call such constraint triple graphs *ground*. We usually depict ground constraints with the attribute values induced by

the formula in the attribute compartments and omit the formula (e.g. see constraint $CTrG$ in Fig. 11). The equivalence between ground constraints and triple graphs is very useful as, from now on, we just need to work with constraint triple graphs. In particular, we are manipulating triple graphs with TGG operational rules, but interpreting triple graphs as ground constraints.

Most of the rules considered in this paper at the operational level are non-deleting (i.e. when applied to a graph they just add to it some new elements). These rules consist of:

- Two constraint triple graphs L and R, where L is embedded in R, which are called the left-hand side (LHS) and the right-hand side (RHS) of the rule, respectively.
- A set of negative pre-conditions, PRE, where each negative pre-condition N_{pre} in PRE is a constraint triple graph that embeds L.
- A set of negative post-conditions, $POST$, where each negative post-condition N_{post} in POST is a constraint triple graph that embeds R.

The application of a rule $\langle L \hookrightarrow R, PRE, POST\rangle$ to a *host* triple graph (seen as a ground constraint triple graph) $CTrG$ can be explained as follows:

- The rule can be applied to $CTrG$ if L can be embedded in $CTrG$, this embedding satisfies the negative pre-conditions in PRE, and the corresponding embedding of R in the result of applying the rule satisfies the negative post-conditions in $POST$.
- The result of the rule application is the pushout $CTrG +_L R$ or, equivalently, the union of R and $CTrG$ taking L as intersection. By construction, R is embedded in the result of the rule application.

In this explanation, we say that an embedding $e : L \hookrightarrow CTrG$ satisfies the pre-condition N_{pre} if it cannot be extended to an embedding $e' : N_{pre} \hookrightarrow CTrG$ (i.e. there is no e' such that e is equal to the embedding $L \hookrightarrow N_{pre} \overset{e'}{\hookrightarrow} CTrG$). Satisfaction of post-conditions is similar.

Example. Fig. 6 shows a derivation example. The rule in the upper part contains two negative pre-conditions, NAC1 and NAC2, the former equal to the RHS. The LHS is embedded in $CTrG$ at two different locations, but we can apply the rule only in the location that is marked with a coloured square, as the other does not satisfy NAC2 (i.e. that embedding can be extended to an embedding of NAC2). Then, the rule is applied in the coloured embedding by performing a pushout: $CTrG$ and the rule's RHS are glued through the rule's LHS. The resulting graph $CTrH$ contains the new created elements, as well as the conjunction of the formulae in $CTrG$ and the rule's RHS. Please note that the rule is no longer applicable in $CTrH$ since we cannot find any embedding of L that satisfies all negative pre-conditions.

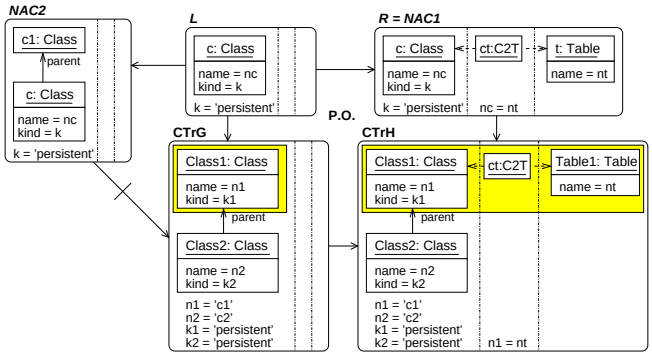

Fig. 6 Derivation example using constraints.

In Section 7.2 we also use deleting rules, which do not create elements but only delete them. These rules are formalized as an embedding $L \hookleftarrow R$, where this time the RHS is embedded in the LHS because the LHS is "bigger". The direct derivation works as before, but with the morphism between $CTrH$ and $CTrG$ reversed, and where the rule effect is the deletion of elements as we calculate the *pushout complement* [2]. See [16] for the details and the formulation of rules that can add and delete in one step. We use these simpler formulations because our rules do not add and delete simultaneously.

Contrary to our approach based on constraint triple graphs, the usual way [16,43] of dealing with triple graphs instead poses some difficulties, most notably concerning attribute handling. For instance, Fig. 7 shows an example where a TGG operational rule is applied to a triple graph G. The rule creates a column for each private attribute starting by '__'. Function `LTRIM(p1,p2)` returns `p2` after removing `p1` from its beginning.

In practice, TGG operational rules like this one are not specified by hand, but derived from declarative rules modelling the synchronized evolution of two models [43], as depicted in the upper part of Fig. 7. The declarative rule is shown with its LHS and RHS together, and *new* tags indicating the synchronously created elements. Attribute computations in the declarative rules must be expressed in a declarative style. However, their compilation into operational rules has to assign a causality to attribute computations, which involves algebraic manipulation of formulae. Moreover, appropriate attribute conditions must be synthesized, too. In the example, the condition *x='__'+y* has to be transformed into a computation *LTRIM('__',x)* for the created column name, and into the condition *x[0:2]='__'* as the attribute name should start by '__'. Unfortunately, this kind of manipulation is difficult to automate, since it involves the synthesis of both operations and conditions. As already seen in Fig. 6, our approach proposes a more straightforward solution. Fig. 16 shows the same example when dealing with triple graphs as ground constraints, where there is

[2] The pushout complement in Fig. 6 calculates $CTrG$ given $CTrH$, R and L, such that the square is a pushout.

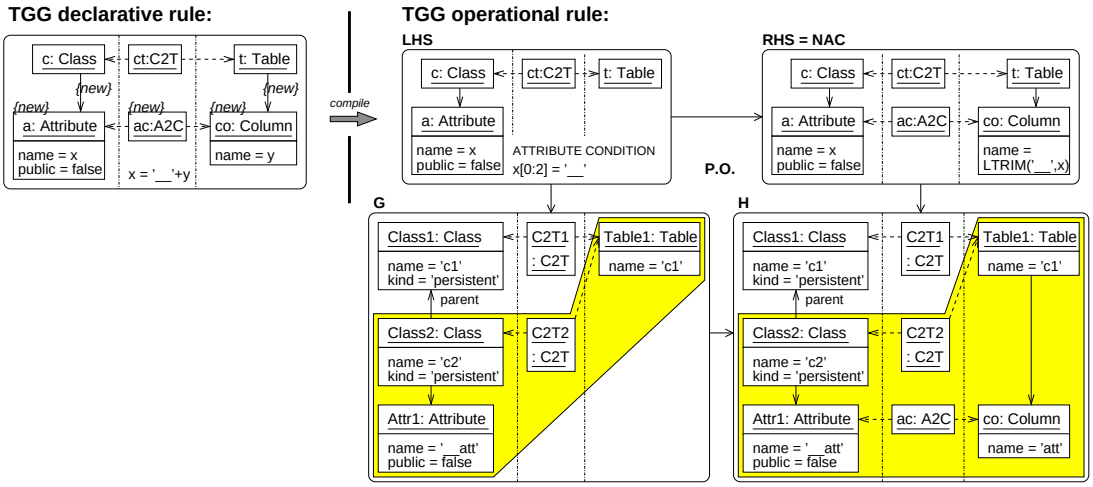

Fig. 7 Direct derivation by a non-deleting TGG operational rule, using graphs instead of constraints.

no need to synthesize attribute computations. The result of a transformation is a pair of models where the attributes are variables with values given by formulae. If needed, a constraint solver can compute concrete values. Moreover, this approach allows for the loose specification of transformations and full exploration of solutions.

Similar to TGGs, our operational rules are also synthesized from a higher-level direction-independent specification. However, we do not use declarative rules, but a new notion of pattern built on top of constraint triple graphs. We present such a notion in the next section.

5 Pattern-Based Inter-Model Specifications

Triple Patterns are similar to graph constraints [16], but made of constraint triple graphs instead of graphs and with a slightly different meaning. We use them to describe the allowed and forbidden relations between two models. Thus, a pattern specification expresses the conditions under which two models are to be considered consistent. We define two kinds of patterns: positive and negative. A positive pattern (P-pattern) describes relations between source and target elements that, under some given circumstances, must be present in any triple graph. A negative pattern (N-pattern) describes relations between source and target elements that should not occur in any triple graph. As the remainder of the paper will show, a pattern specification can be used in different scenarios but, in all of them, we need: (i) to define a suitable notion of *satisfaction* of a specification by models in the particular scenario, and (ii) to obtain low-level operational mechanisms derived from the specification that (re-)establish the consistency in the scenario.

Definition 1 (Triple pattern) *A P-pattern CP, denoted $\bigwedge_{i \in Pre} N(C_i) \wedge C \Rightarrow Q$, consists of:*

- *A constraint triple graph C called the* positive pre-condition *of CP.*
- *A constraint triple graph Q embedding C.*
- *A set $N_{pre}(CP)$ of negative pre-conditions $N(C_i)$, where each C_i is a constraint triple graph embedding Q.*

An N-pattern CP, denoted $N(Q)$, consists just of a constraint triple graph Q.

Remark. The notation $N(\cdot)$ is just syntactic sugar to indicate a constraint interpreted in a negative way. The constraint C is also called *parameter*, while Q is the pattern's main constraint.

The simplest P-pattern is made of a main constraint Q restricted by negative pre-conditions (Pre set) with C empty. In the most common usage scenario for patterns, Q has to be present in a triple graph (i.e. in a ground constraint) whenever the context defined by any negative pre-condition C_i is not found. In this sense, the negative pre-conditions play a similar role as the negative pre-conditions of a graph transformation rule: they denote contexts that, if found, do not require fulfilling the main constraint of the pattern. In this way, if a negative pre-condition is found, it is not mandatory to find Q, but still possible. P-patterns can also have a parameter or positive pre-condition, specified with a non-empty C. In such case, Q has to be found only if C is also found. Finally, an N-pattern is made of a constraint Q which is forbidden to occur.

For simplicity, we do not allow negative pre-conditions restricting the source and the target at the same time. In practice, each negative pre-condition can be split into two pre-conditions: the first one restricting only the source and the second one restricting only the target. This condition has also been imposed by other approaches like [17,18].

Example. Fig. 8 shows the theoretic notation of one pattern, taken from the class to relational schema transformation [40]. The main constraint (named `Class-Table`) specifies that persistent classes should be related to tables with same name. The pattern has

an empty positive pre-condition (named `C`), and a negative pre-condition (named `N(Parent)`) requiring the class to have no parent. Hence, according to Definition 1, `Class-Table` embeds `C`, and `N(Parent)` embeds `Class-Table`. Please note that we model the fact that `Class-Table` has no positive pre-condition by means of an empty parameter C. This is done in order to make uniform the syntax and semantics of all kinds of positive patterns. As we will see later, pre-conditions are a means to declare dependencies between patterns. The semantics of a positive pre-condition will be interpreted differently depending on the concrete inter-modelling scenario, but in general, it expresses certain structure that needs to be present in the models in order to demand the occurrence of the pattern main constraint.

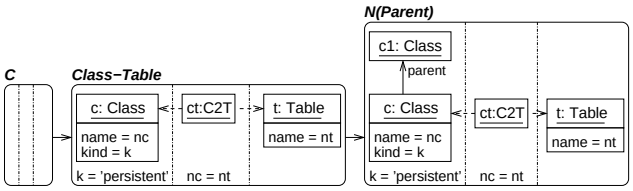

Fig. 8 Theoretic notation for pattern `Class-Table`.

Fig. 9 shows the same pattern, and two additional ones, using a compact, more intuitive notation that we prefer to use. As a shortcut, the negative pre-condition shows only the elements that do not belong to the main constraint, and those connected to them. The two P-patterns below define parameters. More precisely, C consists of the nodes of Q that are labelled with $\langle\langle$`param`$\rangle\rangle$, using a notation similar to stereotypes. As we will see later, positive pre-conditions are a way to encode pattern dependencies. In this way pattern `Parent Class-Table` requires as pre-condition a class and a related table, and such relation will be created by the operational mechanisms generated from both patterns `Class-Table` and `Parent Class-Table`. At the specification level, the P-pattern `Parent Class-Table` expresses that children classes should be mapped to the same table as their parents, whereas the P-pattern `Attribute-Column` maps the attributes of a class with the columns of the table related to the class. In Section 6.3 we will show that, often, it is not necessary to specify parameters, as appropriate heuristics are able to suggest them.

Definition 2 (Inter-model specification) *An inter-model specification $SP = \bigwedge_{i \in I} CP_i$ is a conjunction of patterns, where each CP_i can be positive or negative.*

Next, we introduce the concept of pattern enabledness, a basic concept used to build the notion of pattern satisfaction in different scenarios. Three different kinds of enabledness are defined depending on whether we interpret P-patterns as source-to-target, target-to-source, or source-and-target with respect to the trace. In the first enabling notion, we are interested in checking

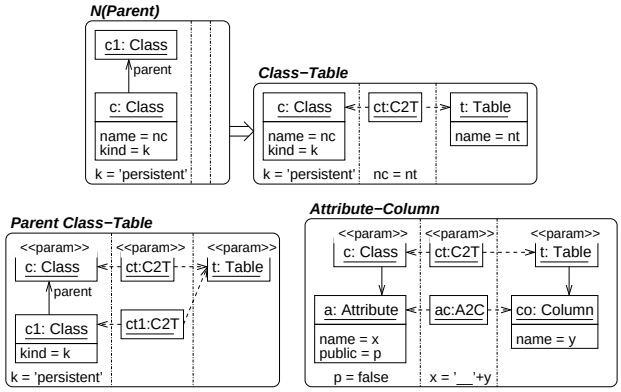

Fig. 9 A pattern-based inter-model specification.

whether the pre-condition of the pattern together with the source of the main constraint is present in the model, without violating the negative pre-conditions of the pattern, in which case we say the pattern is source-enabled. The converse is checked when interpreting the pattern target-to-source. This distinction is useful because, in the forward model transformation scenario, we only demand that whenever a pattern is source-enabled, an occurrence of the main constraint exists (see Section 6). In model matching scenarios we are interested in trace-enabledness, looking simultaneously at both the source and target of the pattern. We do not consider enabledness of N-patterns as their only interpretation is that they are forbidden to occur.

We start by defining the positive and negative forward, backward and trace pre-conditions of P-patterns. Later, we will use these notions to define the three notions of enabledness, and also in later sections to define different notions of satisfaction, and as building blocks for rules.

Definition 3 (Directed pre-conditions) *Given a P-pattern $CP = [\bigwedge_{i \in Pre} N(C_i) \wedge C \Rightarrow Q]$ and a constraint triple graph $CTrG$:*

- *Its positive forward pre-condition is given by $P_S = C +_{C|_S} Q|_S$, i.e. P_S consists of the source part of Q, the trace and target parts of C, and the formula $\alpha^S_Q \wedge \alpha_C$. Its set of negative forward pre-conditions is given by $pre^S(CP) = \{N_i^S = C +_{C|_S} C_i|_S \mid C_i \in N_{Pre}(CP), N_i^S \ncong P_S\}$. This means that for each negative pre-condition C_i in CP, $pre^S(CP)$ includes the negative pre-condition consisting of the source part of C_i, the trace and target parts of C, and the formula $\alpha^S_{C_i} \wedge \alpha_C$ (unless the negative pre-condition coincides with P_S, in which case it is not included in $pre^S(CP)$).*
- *Its positive backward pre-condition is given by $P_T = C +_{C|_T} Q|_T$, i.e. P_T consists of the target part of Q, the trace and source parts of C, and the formula $\alpha^T_Q \wedge \alpha_C$. Its set of negative backward pre-conditions is given by $pre^T(CP) = \{N_i^T = C +_{C|_T} C_i|_T \mid C_i \in N_{Pre}(CP), N_i^T \ncong P_T\}$.*

– *Its positive trace pre-condition is given by* $P_{ST} = C +_{C|_{ST}} Q|_{ST}$*, i.e.* P_{ST} *consists of the source and target parts of* Q*, the trace part of* C*, and the formula* α_Q*. Its set of negative trace pre-conditions is given by* $pre^{ST}(CP) = \{N_i^{ST} = C +_{C|_{ST}} C_i|_{ST} \,|\, C_i \in N_{Pre}(CP), N_i^{ST} \ncong P_{ST}\}$*. This means that for each negative pre-condition* C_i *in* CP*,* $pre^{ST}(CP)$ *includes the negative pre-condition consisting of the source and target parts of* C_i*, the trace part of* C*, and the formula* α_{C_i} *(unless it coincides with* P_{ST}*).*

Remark. By construction, in the three cases P_x is embedded in N_i^x (for $x \in \{\{S\}, \{T\}, \{ST\}\}$).

Example. Fig. 10 shows the forward pre-conditions of the P-pattern `Class-Table` (shown in Fig. 9). The positive one (P_S) demands a persistent class, while the only negative one (N_i^S) asks such class to have no parent. The way of calculating P_S is as follows: we take the pre-condition C (empty) and the source part of the main constraint ($Q|_S$), and do the merging (the pushout) of both through the source restriction $C|_S$ of C. To calculate the negative forward pre-conditions, we take the source restriction of each negative pre-condition in $N_{Pre}(C_i)$ and do the merging with C. The resulting forward pre-conditions state the needed structure in source graphs in order to demand an occurrence of the main constraint Q. Although we do not show it, the positive backward pre-condition is made of one table, and there are no negative backward pre-conditions because the obtained N_i^T coincides with the positive backward pre-condition P_T.

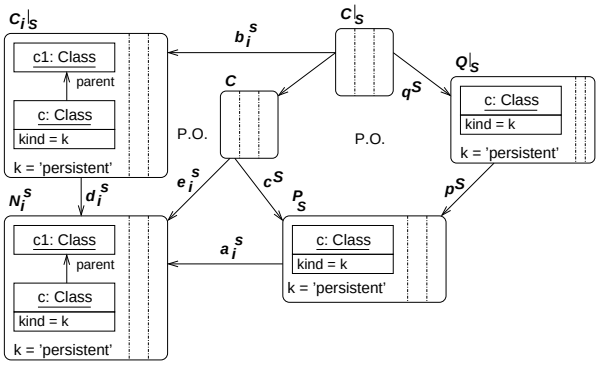

Fig. 10 Forward pre-conditions of pattern `Class-Table`.

Definition 4 (P-pattern enabledness) *Given a P-pattern* $CP = [\bigwedge_{i \in Pre} N(C_i) \wedge C \Rightarrow Q]$ *and a constraint triple graph* $CTrG$*:*

– *CP is source-enabled in* $CTrG$ *at a match* $m^S : P_S \hookrightarrow CTrG$*, written* $CTrG \vdash_{m^S,F} CP$*, if* m^S *is an embedding and for every negative pre-condition* N_i^S *in* $pre^S(CP)$ *there is no embedding of* N_i^S *in* $CTrG$ *extending* m^S*.*

– *CP is target-enabled in* $CTrG$ *at a match* $m^T : P_T \hookrightarrow CTrG$*, written* $CTrG \vdash_{m^T,B} CP$*, if* m^T *is an embedding and for every negative pre-condition* N_i^T *in* $pre^T(CP)$ *there is no embedding of* N_i^T *in* $CTrG$ *extending* m^T*.*

– *CP is trace-enabled in* $CTrG$ *at a match* $m^{ST} : P_{ST} \hookrightarrow CTrG$*, written* $CTrG \vdash_{m^{ST},T} CP$*, if* m^{ST} *is an embedding and for every negative pre-condition* N_i^{ST} *in* $pre^{ST}(CP)$ *there is no embedding of* N_i^{ST} *in* $CTrG$ *extending* m^{ST}*.*

Example. Fig. 11 illustrates the notion of source-enabledness with P-pattern `Class-Table`. There are two embeddings of the pattern's forward pre-condition P_S in $CTrG$. The first one identifies the class c in P_S with the class c in $CTrG$. Such embedding cannot be extended to an embedding of the forward negative pre-condition N_i^S (i.e. c does not have a parent class) and hence the pattern is source-enabled at such embedding. On the contrary, the pattern is not source-enabled at the second match which identifies the class c in P_S with class $c2$ in $CTrG$. The reason is that there is an embedding of the negative forward pre-condition that extends the embedding of P_S, i.e. *c2* does have a parent.

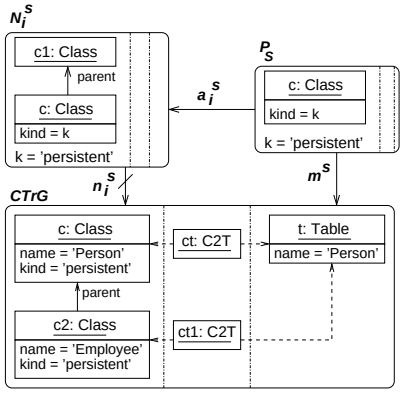

Fig. 11 Source-enabledness of P-pattern `Class-Table` in a constraint triple graph.

As we will see in the following two sections, different scenarios will provide different notions of pattern satisfaction based on the enabledness of patterns (among other conditions). The satisfaction and usage of patterns for specifying M2M transformations is presented next, where forward transformations demand that all source-enabled matches can be extended to the pattern's main constraint. We will show the application of patterns to the rather different scenario of model matching and model traceability in Section 7.

6 Model-to-Model Transformation

In this section we describe the usage of patterns to describe M2M transformations. For this purpose we first define an appropriate notion of pattern satisfaction, namely whether a source and a target models are synchronized according to a pattern specification. Then we

describe how to generate operational rules able to build a target model starting from a source one, or vice versa.

Pattern satisfaction in M2M transformation scenarios can be decomposed in forward and backward satisfaction. The former demands the existence of the main constraint in all places where a pattern is source-enabled. Backward satisfaction demands the same but in all places where a pattern is target-enabled. The separation between forward and backward satisfaction is useful because if we transform forwards (assuming an initial empty target graph) we just need to check forward satisfaction. Full satisfaction – which we call synchronization – implies both forward and backward satisfaction and is only needed to check if two models are actually synchronized.

The next definition formalizes the previous ideas, where we check satisfiability of patterns by constraint triple graphs, which need not be necessarily ground. This is so because, during a transformation, the source and target models do not need to be ground (i.e. they may contain variables that can take several values satisfying the constraints). When the transformation finishes we can use a solver in order to find an attribute assignment satisfying the formulae.

Definition 5 (Synchronization) *A constraint triple graph $CTrG$ is synchronized with respect to a P-pattern $CP = [\bigwedge_{i \in Pre} N(C_i) \wedge C \Rightarrow Q]$, written $CTrG \models CP$, if:*

- *CP is* satisfied forwards*, denoted $CTrG \models_F CP$. This means that whenever CP is source-enabled in $CTrG$ at a match m^S, then there is an embedding $m : Q \hookrightarrow CTrG$ extending m^S, and*
- *CP is* satisfied backwards*, denoted $CTrG \models_B CP$. This means that whenever CP is target-enabled in $CTrG$ at a match m^T, then there is an embedding $m : Q \hookrightarrow CTrG$ extending m^T.*

$CTrG$ is synchronized with respect to an N-pattern $CP = [N(Q)]$, written $CTrG \models CP$, if there is no embedding of Q into $CTrG$.

The forward satisfaction of a P-pattern demands that, for each occurrence of its forward pre-condition P_S, an occurrence of Q must be found containing the pre-condition occurrence. A constraint triple graph $CTrG$ satisfies a P-pattern either because no m^S is found (*trivial satisfaction*), or because m^S and m are found (*positive satisfaction*). Backward satisfaction demands similar conditions. For N-patterns we just demand their absence.

Example. Fig. 12 depicts the forward satisfaction of pattern `Class-Table` by the ground constraint $CTrG$. We have $CTrG \models_F$ `Class` $-$ `Table` because the pattern is source-enabled at just one match, shown by equality of identifiers, and this match can be extended to the main constraint Q. The figure also shows the negative pre-condition, to recall that *c2* does not enable the pattern forwards. Although not shown in the figure, we also have $CTrG \models_B$ `Class` $-$ `Table` because the pattern is target-enabled at just one match that identifies the table in the pattern with the table in $CTrG$, and such a match can be extended to Q. Hence, $CTrG \models$ `Class` $-$ `Table`.

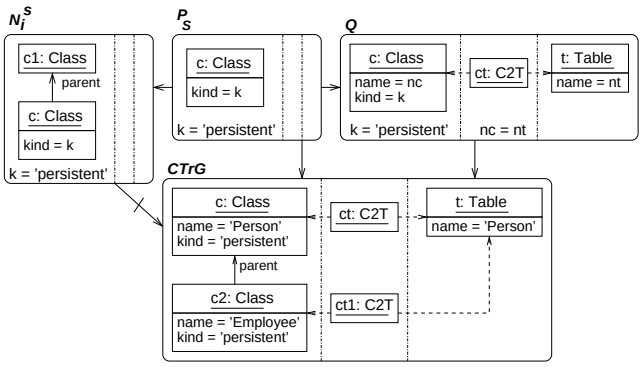

Fig. 12 Forward satisfaction of pattern `Class-Table`.

Given a specification $SP = \bigwedge_{i \in I} CP_i$ and a constraint triple graph $CTrG$, we write $CTrG \models SP$ to denote that $CTrG$ satisfies all patterns in SP. The M2M transformation semantics of a pattern specification is the language of all triple graph constraints (not necessarily ground) that satisfy the specification. This is convenient as, in case of non-ground constraints, a solver can obtain one that is ground and satisfies the specification, if it exists.

Definition 6 (M2M transformation semantics) *Given a specification SP, its M2M transformation semantics $SEM(SP)$ is given by the class of all constraint triple graphs $CTrG$ such that $CTrG \models SP$.*

A constraint $CTrG$ satisfies a specification SP if it satisfies all its patterns (we sometimes say that $CTrG$ is a model of the specification). Hence, any constraint satisfying the specification must belong to the language generated by each pattern CP_i in the specification. Formally, $SEM(SP) = \bigcap_{CP_i \in SP} SEM(CP_i)$. This fact makes compositional the semantics of pattern-based M2M transformations, as adding a new pattern to a specification amounts to intersecting the languages of both. This is useful when extending or reusing specifications.

Proposition 1 (Composition of specifications) *Given specifications SP_1 and SP_2, $SEM(SP_1 \wedge SP_2) = SEM(SP_1) \cap SEM(SP_2)$. (Proof in Appendix).*

6.1 Enriching Specifications with Meta-Model Information

M2M transformation specifications cannot be oblivious to the integrity constraints imposed by the meta-models of the source and target languages. The simplest ones

are the maximum cardinality constraints in association ends. They induce N-patterns that we automatically derive and include in the specifications. This is useful to restrict the number of models that satisfy a specification by ruling out those which are syntactically incorrect.

The generation procedure is simple: if a class `A` is restricted to be connected to a maximum of j objects of class `B`, then we build one N-pattern made of an `A` object connected to $j+1$ `B` objects.

Example. Fig. 13 shows the meta-model triple for the class-to-relational example. As class `Attribute` has to be connected to exactly one `Class`, we generate one N-pattern made of an `Attribute` object connected with two `Class` objects. Similarly, we generate another N-pattern due to the parent association in the source language and the association between `Column` and `Table` in the target.

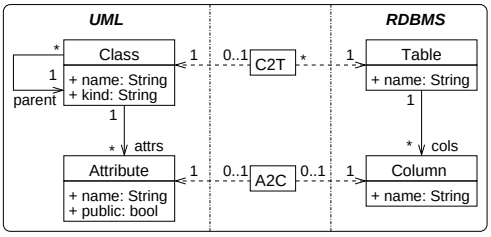

Fig. 13 Simple meta-model triple for UML and RDBMS.

As noted in the meta-model triple, we decorate the correspondence functions with cardinality constraints. These always have to be equal to *1* on the side of the source and target elements, but may vary on the side of the mappings. For instance, classes can receive zero or one mappings, while tables can receive zero or more. The maximum cardinality constraints on the correspondence functions also produce N-patterns, which are shown in Fig. 14. Note that no N-pattern is generated from the cardinalities on the side of the source and target elements as the very formalism takes care of that.

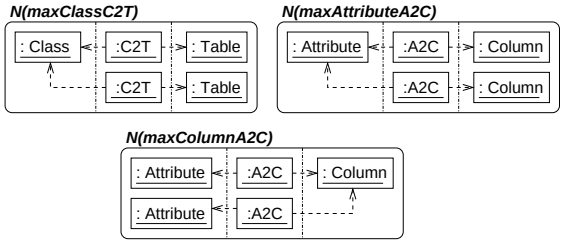

Fig. 14 N-patterns generated from the maximum cardinality constraints in the correspondence functions.

Apart from the maximum cardinality constraints, other restricted forms of OCL (coming from the meta-model) can also be encoded as patterns. For this purpose we can benefit from previous works on translating OCL into graph constraints [47]. Interestingly, once the meta-model constraints are expressed in the form of patterns, we can analyse their consistency regarding the specification. For example, if there is a morphism from some of the generated N-patterns to a P-pattern in the specification, then we can conclude that either the P-pattern is useless as it is subsumed by the N-pattern, or the transformation is incorrect as it could try to create models violating the cardinality constraints.

Inheritance relationships and abstract types in meta-models can also be exploited to define more compact patterns. In particular, objects with abstract typing can appear in patterns, which intuitively is equivalent to the *disjunction* of the n patterns that result from substituting the abstract objects by all its n concrete subtypes.

Example. Just for the sake of illustration, let's suppose that the UML meta-model of our example defines an additional class named `Interface`, and that both `Class` and `Interface` inherit from an abstract class `Container`. In this case, the P-pattern shown in Fig. 15 specifies that both persistent `Class`es and `Interface`s have to be mapped to `Table`s, and that `Table`s have to be mapped to either a `Class` or an `Interface`, hence the disjunction. Note that in addition to making specifications more compact, the fact that using abstract types produces a disjunction of patterns is a way to design loose specifications, to be refined at later stages.

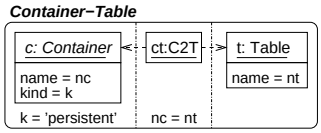

Fig. 15 P-pattern with abstract object.

6.2 Deriving Forward/Backward Transformation Rules

Next we describe the synthesis of TGG operational rules implementing forward and backward transformations from pattern-based specifications. In forward transformation, we start with a constraint triple graph where only the source is present, and the rules build the appropriate correspondence and target models according to the patterns. In backward transformation it is the other way round, we start with just a target and build the source and correspondence. We assume that the initial models do not violate any N-pattern of the specification. As some of these N-patterns are derived from the maximum cardinality constraints of association ends in meta-models, it is reasonable to assume syntactically correct starting models.

The synthesis process derives one rule from each P-pattern, made of triple graph constraints in its LHS and RHS. In particular, the forward pre-condition P_S is

taken as the LHS for the forward rule, and the main constraint Q as the RHS. For simplicity, we neglect abstract typing in this first step. The negative pre-conditions of a P-pattern are used as negative pre-conditions of the synthesized rules. All N-patterns in the specification are converted into negative post-conditions of the rules, using the well-known procedure to convert graph constraints into a rule's post-conditions [16]. Finally, additional NACs are added to ensure termination (see the proof in [38]). For simplicity we only show the generation of forward rules, as backward rules are generated analogously [25].

Before stating this definition, let us explain an overlapping construction that we need below. In particular, we say that a constraint triple graph $CTrG$ is the overlapping of $CTrG_1$ and $CTrG_2$ if $CTrG$ embeds $CTrG_1$ and $CTrG_2$ via e_1 and e_2 (i.e. $e_1 : CTrG_1 \hookrightarrow CTrG$ and $e_2 : CTrG_2 \hookrightarrow CTrG$) and e_1 and e_2 are jointly surjective, which means that every element (node or edge) in $CTrG$ comes from $CTrG_1$ or $CTrG_2$. That is, we can consider that $CTrG$ is a kind of union of $CTrG_1$ and $CTrG_2$. Moreover, we say that $CTrG$ is the overlapping of $CTrG_1$ and $CTrG_2$ with respect to $CTrG_0$ if, additionally, $CTrG_0$ is embedded in the intersection of e_1 and e_2). Notice that this is different from a pushout: if $CTrG = CTrG_1 +_{CTrG_0} CTrG_2$, then $CTrG$ can be seen as the union of $CTrG_1$ and $CTrG_2$, where $CTrG_0$ coincides with the intersection; however, in an overlapping, we only require that $CTrG_0$ is embedded in the intersection.

Definition 7 (Derived forward rule) *Given a specification SP and a P-pattern $CP = [\bigwedge_{i \in Pre} N(C_i) \wedge C \Rightarrow Q] \in SP$, the following forward rule is generated:*

$$\overrightarrow{r_{CP}} = (P_S \xrightarrow{r} Q, pre(CP), post(CP))$$

where:

- *$P_S \xrightarrow{r} Q$ is the main part of the rule, where P_S is the LHS and Q the RHS.*
- *The set $pre(CP)$ of negative pre-conditions is defined as $pre(CP) = pre^S(CP) \cup TNAC(CP)$, where $pre^S(CP)$ is the set of negative forward pre-conditions (see Definition 3), and $TNAC(CP)$ is a set of NACs ensuring termination, consisting of all constraint triple graphs T which are an overlapping of Q and P_S with respect to $Q|_S$.*
- *The set $post(CP)$ consists of all constraint triple graphs D which are an overlapping of Q and Q', for some N-pattern $N(Q')$ in SP, and such that $(Q \setminus P_S) \cap Q'$ is not empty.*

The set $post(CP)$ contains the rule's negative post-conditions derived from the N-patterns of the specification. This is done by overlapping each N-pattern with the rule's RHS in all possible ways. Moreover, the requirement that $(Q \setminus P_S) \cap Q' \neq \emptyset$ reduces the size of $post(CP)$ because we only need to consider possible violations of the N-pattern due to elements created by the RHS.

Example. The upper row of Fig. 16 shows the operational forward rule generated from pattern `Attribute-Column`. The set pre^S is empty as the pattern has no negative pre-conditions. There are two NACs for termination, `TNAC1` and `TNAC2`, the former equal to R. They are the two different ways of overlapping L and R with respect to $R|_S$. While `TNAC1` identifies the `C2T` and the `Table` from L and R, `TNAC2` does not identify them[3]. As a difference from Fig. 7, we do not need to do algebraic manipulation of formulae to generate the rule, demonstrating the advantage of our way of directly handling attribute conditions. The figure also shows a direct derivation where both G and H are ground constraints. Note that we do not check in L that x starts with "_ _", but if it does not, we would obtain an unsatisfiable constraint.

If a rule creates objects having a type with defined subtypes, we generate a set of rules resulting from substituting the type by all its concrete subtypes. This substitution is not necessary in the elements of the LHS as they are not created, and it is not done in the NACs in order to obtain the expected behaviour of disjunction. Using an optimization similar to [10] one could work directly with abstract rules, but we would have to modify the notion of embedding, which is left for future work.

Example. Fig. 17 shows the backward rules generated from the pattern with abstract object `Container-Table`, shown in Fig. 15. As the backward rules create the `Container`, and this has two concrete children classes, two rules are generated that create either a `Class` or an `Interface`. Each rule has a termination NAC, where the `Container` object has not been substituted by an object with concrete type. This ensures that the `Table` is connected to either a `Class` or an `Interface`. Note that only one forward rule is generated from the pattern because the `Container` is not created in forward transformation. Hence, given an initial `Table`, these rules will generate two admissible solutions according to the specification.

According to [38], the generated rules are terminating and, in absence of N-patterns, correct: they produce only valid models of the specification. However, the rules are not complete: not all models satisfying the specification can be produced by the rules. The next subsection describes a method, called *parameterization*, which in addition ensures completeness of the rules generated from a specification without N-patterns.

If a specification contains N-patterns, these are added as negative post-conditions to the rules generated from the P-patterns, preventing the occurrence of N-patterns

[3] There is a third way of overlapping, where the `Table`s in L and R are identified together but not the `C2T`. We have simplified this case, as it is subsumed by `TNAC1` (if `TNAC1` is found in a graph, so will be this third overlapping, which becomes useless).

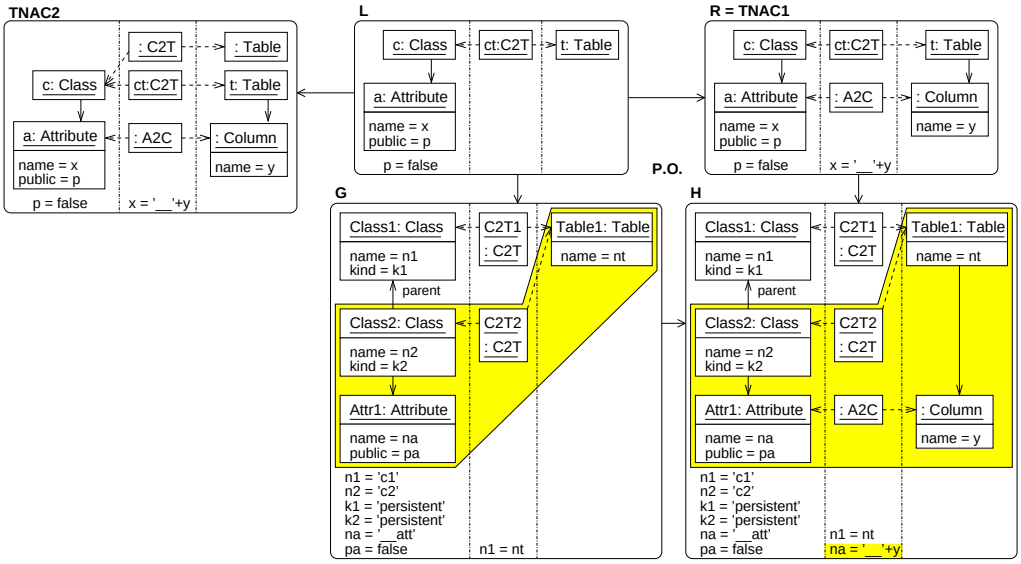

Fig. 16 Forward rule generated from pattern `Attribute-Column`, and direct derivation.

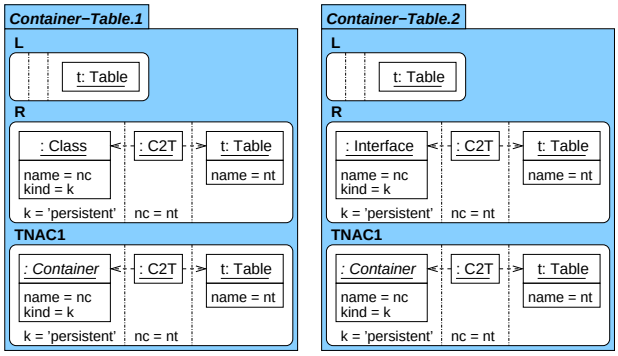

Fig. 17 Backward rules for pattern `Container-Table`.

in the model. However, they may forbid applying some rules before a valid model is found, thus producing graphs that may not satisfy all P-patterns (because the transformation stopped too soon). In this situation the operational mechanism would not be able to find a model, even if it exists. The next subsection presents one heuristic that ensures finding models, and hence correctness, for mechanisms derived from specifications with only certain classes of N-patterns.

6.3 Parameterization and Heuristics in Rule Derivation

Applying the *parameterization* operation to each P-pattern in the specification ensures completeness of the operational mechanism: the rules are able to generate all possible models of the specification [38]. The operation takes a P-pattern and generates additional ones, with all possible positive pre-conditions "bigger" than the original pre-condition, and "smaller" than the main constraint Q. This allows the rules generated from the patterns to reuse already created elements.

Definition 8 (Parameterization) *Given a P-pattern* $CP = [\bigwedge_{i\in Pre} N(C_i) \wedge C \Rightarrow Q]$*, its parameterization is* $Par(CP) = \{\bigwedge_{i\in Pre} N(C_i) \wedge C' \Rightarrow Q \mid C \overset{i_1}{\hookrightarrow} C' \overset{i_2}{\hookrightarrow} Q, C \ncong C', C' \ncong Q\}$.

Remark. The formula $\alpha_{C'}$ can be taken as the conjunction of α_C for the variables already present in ν_C, and α_Q for the variables not in ν_C (i.e. in $\nu'_C \setminus i_1(\nu_C)$). Formally, $\alpha_{C'} = \alpha_C \wedge \alpha_Q|_{i_2(\nu_{C'} \setminus i_1(\nu_C))}$ (assuming no renaming of variables).

Example. Fig. 18 shows some of the parameters generated by parameterization for a pattern like `Attribute-Column` in Fig. 9 but without parameters. Parameterization generates 45 patterns in total. The new pattern with parameter *1* is enforced when the class is already mapped to a table, and in forward transformation avoids generating a rule that creates a table with arbitrary name. Parameter *3* reuses a column with the same name as the attribute (but starting by '__'), possibly created by a parent class. However, parameter *2* is potentially harmful as it may lead to reusing a column connected to a different table, and thus to an incorrect model. Nevertheless, in this case, the N-patterns generated from the maximum cardinality constraints in the meta-models would not allow a column to be connected to more than one table (see Section 6.1).

As the example shows, parameterization generates an exponential number of patterns with increasingly bigger parameters, and therefore an exponential number of derived rules. However, one does not need to generate these rules beforehand, but they can be synthesized "on the fly". That is, we can generate the rule with smallest LHS, and a matching mechanism could try to extend matches of such LHS to bigger subgraphs, but smaller than the RHS. Also note that some of the generated rules will be equal, namely those generated from parameters with the same target and correspondence graph in the case of

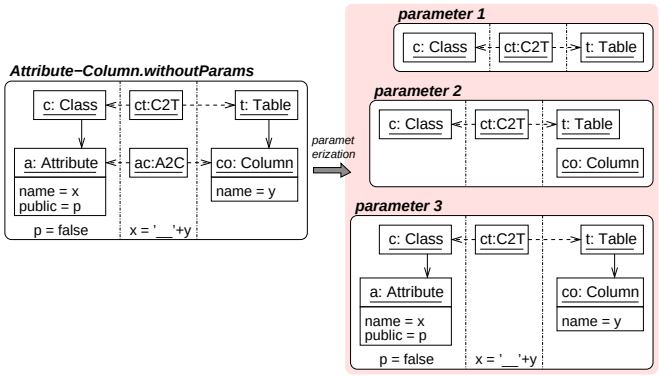

Fig. 18 Parameterization example.

forward rules. Although parameterization ensures completeness [38], we hardly use it in practice but prefer using heuristics to generate just the strictly necessary parameters. However, as previously stated, generating fewer patterns can make the rules unable to find certain models of the specifications (those "too small").

In order to reduce the number of patterns (and consequently of rules) we propose two heuristics. The first one is used to derive only those parameters that avoid the creation of elements with unconstrained attribute values. The objective is to avoid – whenever possible – synthesizing rules creating elements whose attributes can take several values. Thus, we prefer that these elements are generated by some other rule that assigns them a value, if it exists.

Heuristic 1 *Given a P-pattern CP, replace it by a new pattern that has as parameters all elements with some attribute not constrained by the formula in CP but constrained by some other pattern, as well as the mappings and edges between these elements. We do not apply the heuristic if the obtained parameter is equal to Q.*

Example. In Fig. 18, the heuristic generates just one pattern with parameter *1*, which replaces the original pattern. Thus, the generated forward rule avoids creating a table with arbitrary name, whereas the backward one prevents the creation of classes with arbitrary name.

The next heuristic generates only those parameters that avoid duplicating a graph S_1 whenever there is some N-pattern of the form $N(S_1 +_U S_1)$ forbidding the duplication of S_1. This ensures the generation of rules producing valid models for the class of specifications with N-patterns of this form (called FIP in [11]).

Heuristic 2 *Given a P-pattern $CP = [\bigwedge_{i \in Pre} N(C_i) \wedge C \Rightarrow Q] \in SP$, if there is an N-pattern $[N(S)] \in SP$ with $S \cong S_1 +_U S_1$, and such that U is embedded in C, and S_1 is embedded in Q but not in C, then we generate additional patterns having as parameters all C'_j such that C'_j is the overlapping of Q and S_1 with respect to U.*

The rationale of this heuristic is that if a P-pattern has a parameter C that contains U but not S_1, and its main constraint Q contains S_1, then applying the pattern creates a new structure S_1 glued to an existing occurrence of U. This heuristic enlarges the parameter to include S_1 and thus avoids its duplication as occurrences of S_1 will be reused whenever they exist. The way to proceed is to apply the heuristic for each P- and N-pattern of the form $N(S_1 +_U S_1)$, and repeat the procedure with the resulting patterns until no more different patterns are generated.

Example. Fig. 19 shows the application of heuristic 2 to `N(maxParent)` and a variation of the P-pattern `Parent Class-Table`. The former was generated from the maximum cardinality constraint in the association end *parent*. The variation of the P-pattern has as parameter the child class instead of the parent one, just for the sake of illustration. The N-pattern can be decomposed in two isomorphic graphs S_1 with intersection U, and there are embeddings from U to the P-pattern parameter C, and from S_1 to the P-pattern main constraint Q (but not to C). Hence the patterns are suitable for the heuristic, which adds the new pattern $C' \rightarrow Q$ to the specification, maintaining also the original one. The parameter C' is the glueing of C and S_1 through U, and it forbids generating a new parent class in backward transformation if it already exists.

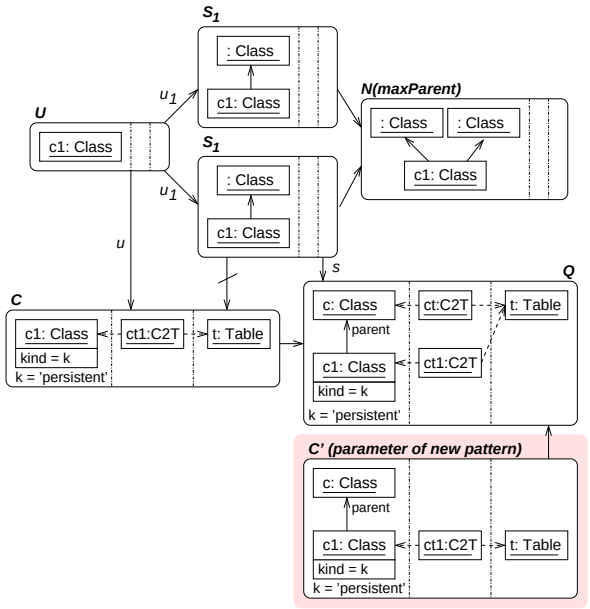

Fig. 19 Heuristic 2 example.

To sum up, next we enumerate the different pattern usage procedures depending on the operations and heuristics presented so far. In particular, when using a pattern-based specification on a M2M transformation scenario, we have the following three options:

– do not apply parameterization or heuristics. In such a case parameters in patterns have to be manually specified, which is more time consuming but allows

for a more refined control of the transformation behaviour. Also, without parameterization, we do not have completeness, although this can be unimportant for specific kinds of transformations.
- apply parameterization and no heuristic. We gain completeness of the operational mechanism, and in some occasions we can save from manually specifying all or some parameters in our patterns. The disadvantage is the exponential number of generated rules, which requires combining this approach with the use of heuristics.
- apply heuristics. The number of generated rules is drastically reduced, as many meaningless parameters are not generated. After applying heuristics it is possible to decide whether we are interested in completeness of the new optimized specification, and then applying parameterization to those patterns.

None of the previous usage procedures is better than the other, as different types of transformations may require a different strategy. A practical example of the use of patterns is shown in Section 8.

7 Model Matching and Model Traceability

In this section we show the usage of patterns for model matching and traceability. The purpose of both activities is generating appropriate traces between the elements of two models, as well as checking whether two models are correctly traced according to a specification. However they differ in the notion of "correct trace". In model matching [29] one pursues the generation of traces between all elements considered similar by the specification. This can be a previous step for further activities like model merging. In model traceability [27] the concern is to establish, maintain and check the correctness of traceability relations between models.

Fig. 20 illustrates the difference between both activities in the context of our running example. The initial models to be traced are shown above. The left one contains two classes and the right one two tables, all with the same name. The model matching scenario generates four traces making explicit the fact that each class is considered similar to any of the two tables, since they have the same name (for illustrative purposes we are neglecting the N-pattern that forbids connecting a class with two tables). On the contrary, the traceability scenario matches exactly one class with one table, obtaining the two solutions shown to the right of the figure. One can interpret each solution as a possible result of either a forward or a backward transformation, but in which the source and target models are not modified and only their traces are established. Thus, a forward transformation would generate a table and a trace from each class, but it would not generate the four traces due to the termination conditions of our rules. Whereas the solutions obtained by model traceability can be generated either by forward or backward transformation, in general, the model matching result can not. Interestingly, the traces generated by model matching are the union of all the traces generated in the different solutions computed by model traceability.

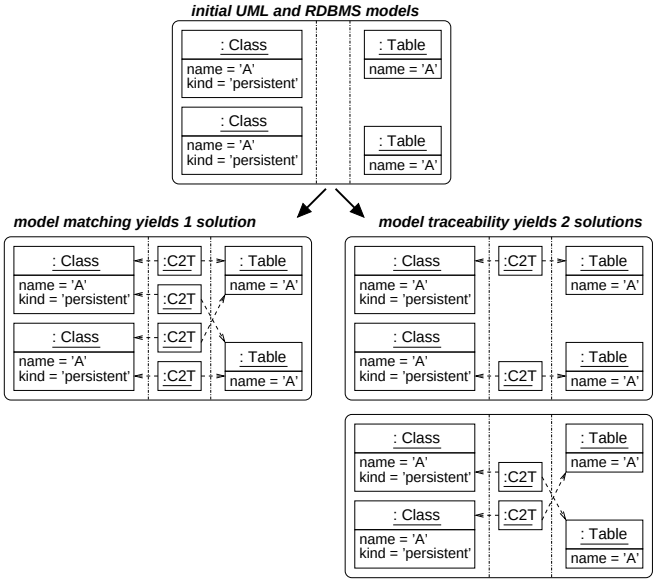

Fig. 20 Model matching and model traceability scenarios.

Next we introduce two notions of pattern satisfaction for model matching and traceability, which tell us whether two models are matched or traced correctly according to a pattern. Unlike the synchronization notion for M2M transformation in Definition 5, these two notions are symmetric in the sense that they look at the source and target elements at the same time and then check for the existence of traces, instead of looking at patterns source-to-target or target-to-source. The notion of satisfaction for N-patterns remains unaltered as they are not interpreted forwards or backwards.

We start with the notion of satisfaction for model matching. This demands that, for each combination of the source and target model elements where a pattern is trace-enabled (recall Definition 4), there is an occurrence of the pattern's main constraint.

Definition 9 (Matching) *A constraint triple graph $CTrG$ is matched with respect to a P-pattern $CP = [\bigwedge_{i \in Pre} N(C_i) \wedge C \Rightarrow Q]$, written $CTrG \models^M CP$, if whenever CP is trace-enabled at a match m^{ST} (i.e. $CTrG \vdash_{m^{ST},T} CP$) there exists an embedding $m : Q \to CTrG$ extending m^{ST}.*

A constraint triple graph $CTrG$ is matched with respect to an N-pattern $CP = [N(Q)]$, written $CTrG \models^M CP$, if Q is not embedded in $CTrG$.

Example. Fig. 21 shows a constraint $CTrG$ that is correctly matched with respect to the pattern `Class-Table`. The notion of matching demands that for each occurrence of P_{ST} containing the source and target of the

pattern (i.e. for each persistent class and table with the same name) a trace exists. There are four such occurrences in the constraint, and hence four traces are demanded. The constraint is also correctly matched with respect to the pattern `Attribute-Column` as this pattern is not trace-enabled in the constraint (i.e. the table *t1* in the constraint should have defined a column with name *'Age'* for the pattern to be trace-enabled). Indeed, the fact that a constraint is correctly matched does not mean that the models it contains are synchronized in the sense of M2M transformation. Our use of patterns for model matching (and traceability) is flexible enough to build trace models on partial models. An example of non-matched constraint with respect to pattern `Class-Table` is $CTrG$ in Fig. 22, as two traces are missing.

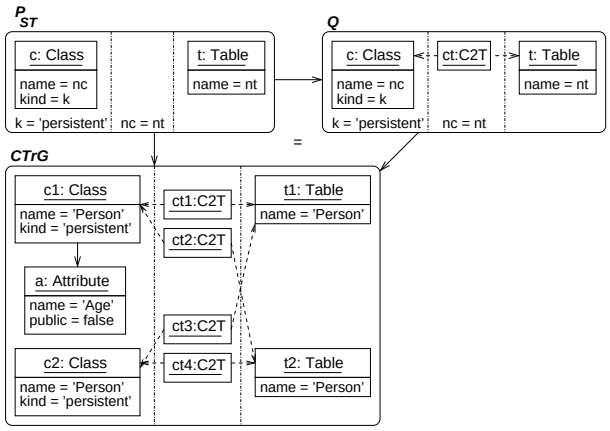

Fig. 21 Constraint matched according to pattern `Class-Table`.

The notion of pattern satisfaction in model traceability is weaker than the one for matching. While the latter demands a universal existence of traces (i.e. for all possible combinations of source and target elements), traceability demands them existentially (i.e. for at least one of such occurrences). Thus, satisfaction in model traceability requires that, for each combination of the source and target model elements where a pattern is trace-enabled, there is an occurrence of the pattern's main constraint that includes the source or the target elements. If there is no trace between a certain occurrence of the source and a certain occurrence of the target, it is because one of them is traced with a different occurrence.

There is an additional detail though: we demand a "uniform" distribution of traces, so that it is not allowed having one occurrence of the source to be traced twice, whereas another occurrence that could have been related with the same target elements as the first one is not traced at all (and similarly for the target). Otherwise, this would mean that the first occurrence has a "redundant trace" that could have been used to connect the second occurrence. Hence, the source part of the trace-enabled occurrences must be traced unless there are not enough target elements (and vice versa). This agrees with forward/backward transformation, which does not enforce a pattern twice in the same set of source/target elements. Put in other words, one can think that our notion of traceability demands traces between the maximal subsets of source and target elements that are synchronized. As a source and a target graphs can be synchronized in different ways, traceability yields a different solution for each one of these ways. Traceability is therefore an important first step towards synchronizing to initially unrelated models: first, traces are established between elements that are already synchronized, and then repairing actions can be performed over elements that are not traced.

Altogether, the definition of pattern satisfaction in model traceability is a bit involved. To make it easier to understand let us first define some terminology. Let us assume that Q is a constraint triple graph. If $e : Q|_S \hookrightarrow CTrG$ (resp. $e : Q|_T \hookrightarrow CTrG$) is an embedding, then we say that e *is traced* by Q if there is an embedding $m : Q \hookrightarrow CTrG$ such that $e = m^S$ (resp. $e = m^T$). Similarly, we say that e *is not traced* by Q if the converse property holds. We also say that e *is doubly traced* by Q if there are two different embeddings $m : Q \hookrightarrow CTrG$ and $m' : Q \hookrightarrow CTrG$ such that $m^S = e = m'^S$ (resp. $m^T = e = m'^T$). Finally, given an embedding m^{ST} from P_{ST} into $CTrG$, we write m^S for the embedding of the source part of P_{ST} (i.e. P_S) in $CTrG$, and similarly for m^T.

Definition 10 (Traceability) *A constraint triple graph $CTrG$ is traced with respect to a P-pattern $CP = [\bigwedge_{i \in Pre} N(C_i) \wedge C \Rightarrow Q]$, written $CTrG \models^T CP$, if whenever CP is trace-enabled at a match m^{ST}, i.e. $CTrG \vdash_{m^{ST},T} CP$, the following two conditions are satisfied:*

1. *If m^S is not traced by Q then m^T is traced, but if there is a different embedding of P_S then this is not doubly traced by Q.*
2. *If m^T is not traced by Q then m^S is traced, but if there is a different embedding of P_T then this is not doubly traced by Q.*

A constraint triple graph $CTrG$ is traced with respect to an N-pattern $CP = [N(Q)]$, written $CTrG \models^T CP$, if Q is not embedded in $CTrG$.

Remark. Condition 1 states that, for each combination of source and target elements that could be matched (i.e. for each occurrence of P_{ST}), if the source part P_S is not traced, then the target part must be traced. Moreover, if P_S is not traced, we forbid any other occurrence of the source to be traced with two occurrences of the target that could have been matched with P_S. Condition 2 demands the reverse situation.

Example. Fig. 22 shows a constraint $CTrG$ that is correctly traced with respect to the pattern `Class-Table`.

The traceability conditions demand that, for each occurrence of P_{ST}, we find an occurrence of the pattern's main constraint Q commuting with the source or target parts of P_{ST}. That is to say, for each persistent class and table with the same name, the class must be related to a table with equal name, or the table must be related to a persistent class with equal name, or the class and the table must be actually related. The figure depicts such occurrences by using inscriptions in the embedding arrows. For example, the occurrence of P_{ST} that contains *c1* and *t1* satisfies the traceability conditions as there is a trace relating both elements. The occurrence of P_{ST} that contains *c1* and *t2* also satisfies the traceability conditions as *c1* and *t2* belong to different occurrences of Q. In fact, finding just one of such occurrences is enough to satisfy the traceability conditions.

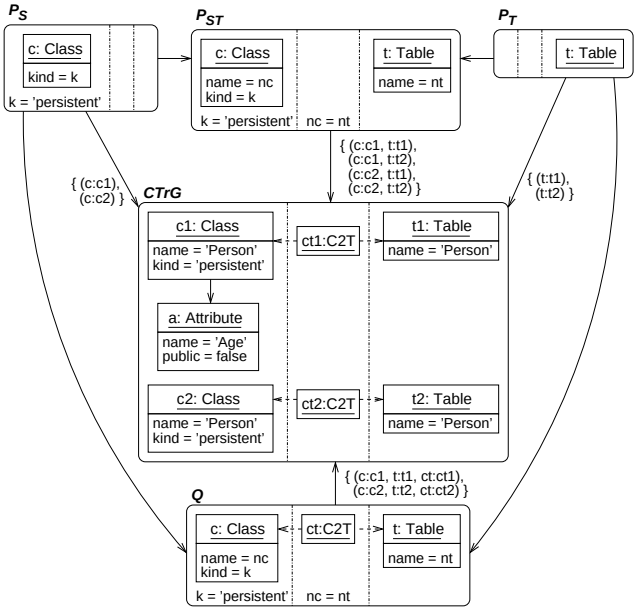

Fig. 22 Constraint traced according to pattern `Class-Table`.

Example. Fig. 23 shows a constraint that is not correctly traced. Although for each combination of class and table (P_{ST}) at least one of them satisfies the pattern, the lower class is not traced at all whereas the upper class (another embedding of P_S) is doubly traced hence "stealing" the trace from the lower class.

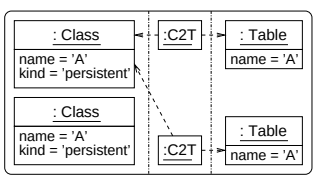

Fig. 23 Invalid traced constraint.

Altogether, a constraint that is matched with respect to a pattern is also traced with respect to the same pattern. However the converse does not necessarily hold. Similarly, two models synchronized as in Definition 5 are correctly traced as well, but not necessarily matched. Finally, two matched models may not be synchronized since we can match non-synchronized models (e.g. a class diagram with an attribute for which there is no column). The relations between the three notions of satisfaction presented so far are summarized in the following proposition.

Proposition 2 (Satisfaction relationships) *Given a specification SP and a constraint triple graph $CTrG$, the following relationships hold: $CTrG \models SP \Rightarrow CTrG \models^T SP$ and $CTrG \models^M SP \Rightarrow CTrG \models^T SP$. (Proof in Appendix)*

7.1 Deriving Relating Rules

Next we provide the generation mechanism for the operational rules that create the traces between two models in the model matching and traceability scenarios (we call them relating rules). In both cases the mechanism derives one rule from each P-pattern; the difference is the generated set of NACs for termination: in *matching* it is just the RHS, whereas in *tracing* it is the union of the forward and backward termination NACs built as in Definition 7.

Definition 11 (Derived relating rules) *Given a specification SP and a P-pattern $CP = [\bigwedge_{i \in Pre} N(C_i) \wedge C \Rightarrow Q] \in SP$, we derive the following relating rule:*

$$rr_{CP} : (P_{ST} \xrightarrow{r} Q, pre(CP), post(CP))$$

where:

- *$P_{ST} \rightarrow Q$ is the main part of the rule, where the positive trace pre-condition P_{ST} is the LHS.*
- *The set $pre(CP)$ of negative pre-conditions is defined as $pre(CP) = pre^{ST}(CP) \cup TNAC(CP)$, where $pre^{ST}(CP)$ is the set of negative trace pre-conditions (see Definition 3), and $TNAC(CP)$ is the set of NACs ensuring termination. For the case of matching rules, it contains just one NAC equal to the rule's RHS. For the case of traceability rules, $TNAC(CP) = TNAC^S(CP) \cup TNAC^T(CP)$ is the union of the sets of forward and backward termination NACs as described in Definition 7.*
- *The set $post(CP)$ is built as in Definition 7.*

Remark. The sets $TNAC^S(CP)$, $TNAC^T(CP)$ and $post(CP)$ are built as in Definition 7, but the LHS and $pre^{ST}(CP)$ consider the source and target of patterns instead of only one of them. Besides, we do not concretize abstract types because the rules do not create elements with those types but just traces. In the traceability rules, the set $TNAC^S(CP)$ forbids tracing twice

the same set of source elements. This agrees with the generated forward transformation rules of Definition 7, which do not enforce a pattern twice in the same set of source elements, and also with the conditions for not duplicating traces of Definition 10.

Example. Fig. 24 shows above the model matching rule derived from pattern `Class-Table`. The rule creates a trace between a class and a table whenever the class is persistent, it has the same name as the table, and it has no parents (as specified by the `NAC1` that comes from the negative pre-condition of the pattern). The rule has one termination NAC, equal to the RHS, preventing its application more than once in the same location. Below, the figure shows the traceability rule generated from the same pattern, which is equal to the matching rule except that it contains two additional termination NACs: $\texttt{TNAC2}^S$ forbids applying the rule if the class is already connected to a table, and $\texttt{TNAC2}^T$ forbids it if the table is connected to a class. Note that, differently from forward and backward rules, applying a relating rule does not add new formulae to the host model.

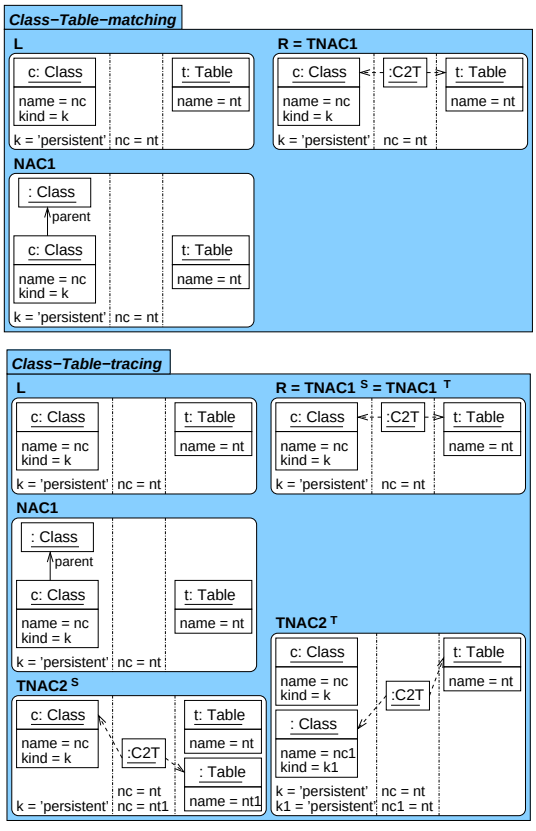

Fig. 24 Relating rules derived from pattern `Class-Table`.

As in the case of M2M transformation, we can apply parameterization and heuristics prior to rule generation with the same trade-offs. Nonetheless, relating rules have the source and target of patterns as their LHS, therefore the number of possible parameters that give rise to different rules is considerably lower. In particular, a pattern whose main constraint has n traces produces at most $2^n - 1$ different rules, but it should be taken into account that patterns rarely contain more than two or three traces. In this sense, we can optimize parameterization by generating only those parameters that differ in their traces.

Also, as in the M2M transformation scenario, we can enrich specifications with N-patterns derived from the meta-model and, in particular, from the cardinality annotations in the correspondence functions (see Section 6.1 and Fig 14). However, we do not consider such N-patterns for model matching as here the aim is to relate source and target elements in all possible ways.

We are aware that in general scenarios like requirements traceability [35] it is difficult to create automatically the traces between analysis and design models from scratch. In practice these traces are manually created by engineers. However, one still may have a specification describing inter-model consistency conditions and use the operational mechanisms to keep the traces consistent when the models change (see the next section). For other scenarios that are amenable to complete automation, like relating a class diagram to a relational model, the operational rules for traceability create the correct traces and return all possible solutions. Furthermore, these rules are useful as a first step towards establishing synchronization starting from two unrelated models.

7.2 Handling Incorrect Traces

The notions of correctly related models presented so far make sure that the needed traces exist, but do not guarantee the absence of incorrect traces. This is so because all presented notions of satisfaction iterate on occurrences of the source and target elements and check the existence of an appropriate trace, but do not iterate on the occurrences of traces checking their validity. Hence, even two models synchronized with respect to a specification may have incorrect traces (in addition to the correct ones) if somebody manually added an incorrect trace relating them, or if the models evolved so that some traces became incorrect. In this case we make a closed world assumption: only those traces that are correct according to the specification should exist.

This section shows how to generate rules able to detect and delete incorrect traces. We assume that the trace model is a discrete graph (i.e. it contains no edges connecting traces with each other). The generated rules must check that, whenever there is a trace, it is because some P-pattern demands its presence and it does not belong to an occurrence of any N-pattern. However, the rules deleting such incorrect trace cannot have just one trace node in its LHS as our formalization requires each trace to relate source and target elements. Hence,

their LHS contains so-called trace triples which are made of a trace node connecting source and target elements. Given a meta-model triple we derive a set Trc of trace triples consisting of all different triple graph constraints typed by the meta-model triple, with one object in the source, target and correspondence graphs (and the formula equal to **true**).

Example. Fig. 25 shows the set of trace triples derived from the meta-model in Fig. 13. The set contains two traces, which correspond to the mappings between `Class`es and `Table`s and between `Attribute`s and `Column`s. As the attributes of the source and target elements are not constrained by the formula, we omit them.

Fig. 25 Trace triples for the meta-model triple in Fig. 13.

Next we identify the set of enabling P-patterns that demand the existence of a certain trace triple, and the set of disabling N-patterns that contain a certain trace type.

Definition 12 (Enabling and disabling patterns) *Given a specification $SP = \bigwedge_{k \in K} CP^k$ and a trace triple $t_i \in Trc$:*

- *The set of enabling patterns of t_i, $Patt^+(t_i)$, is the set of all embeddings $q_i^k : t_i \hookrightarrow Q^k$ such that q_i^k is not the extension of an embedding of t_i in C^k, where C^k and Q^k are the positive pre-condition and main constraint of the P-pattern CP^k.*
- *The set of disabling patterns of t_i, $Patt^-(t_i)$, is the set of all embeddings $q_i^k : t_i \hookrightarrow Q^k$ such that CP^k is the N-pattern $N(Q^k)$.*

Example. Fig. 26 shows the trace triple t_i for the mappings between classes and tables. It has two enabling patterns: `Class-Table` with main constraint Q^1, and `Parent Class-Table` with main constraint Q^2. The class in t_i is related to c in Q^2 but not to $c1$ because $c1$ is in the pattern's parameter. Fig. 27 shows the disabling patterns for the same trace triple in a traceability scenario where `N(maxClassC2T)` is part of the specification (recall that this N-pattern was generated from the cardinality annotations of the correspondence function, and that it is not generated for model matching). There are two elements in the set of disabling patterns because there are two ways of embedding the triple in the pattern, and the set $Patt^-(t_i)$ contains embeddings.

We can check trace correctness at two levels. At the *relaxed* level, we check that any trace triple t_i relating two models actually connects model fragments according to some t_i's enabling pattern. This notion of correctness does not take into account the negative pre-conditions of patterns, since a pattern with negative pre-conditions specifies what should happen if the negative

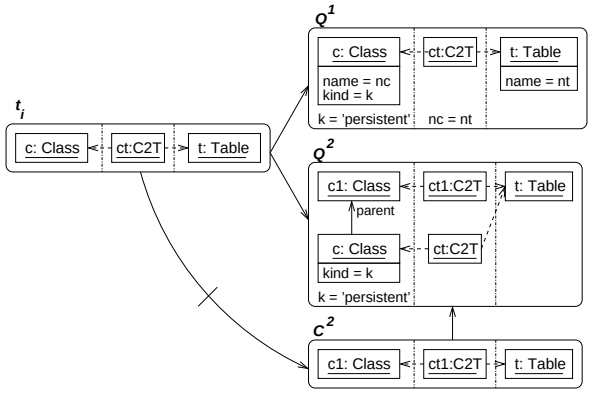

Fig. 26 Enabling patterns set for trace triple.

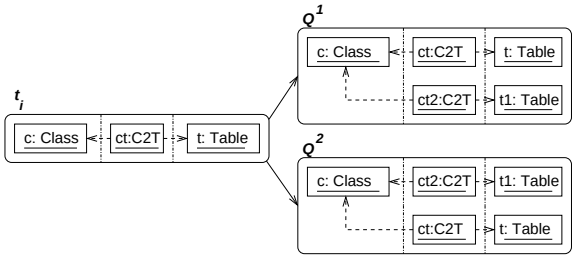

Fig. 27 Disabling patterns set for trace triple.

pre-conditions are not found, but not if they are found. Note however that our (forward, backward, matching and traceability) operational rules include NACs that forbid enforcing a pattern if the negative pre-conditions are found. Therefore, at the *strict* level, we check that only those traces that our operational rules are able to create actually exist. Thus, if there is an occurrence of a trace triple between two related models, then such occurrence should belong to an occurrence of some t_i's enabling pattern for which its negative pre-conditions are not found. These two notions are formalized next.

Definition 13 (No incorrect traces) *Given a specification $SP = \bigwedge_{k \in K} CP^k$, we say that a constraint triple graph $CTrG$ has no incorrect traces at* relaxed level *with respect to SP if for every trace t in Trc, if $m : t \hookrightarrow CTrG$ is an embedding then the following two conditions must be satisfied:*

1. *t is enabled by some positive pattern $CP = [\bigwedge_{i \in Pre} N(C_i) \wedge C \Rightarrow Q]$ (i.e. $m' : t \hookrightarrow Q$ is in $Patt^+(t)$) and there exists an embedding m'' of Q in $CTrG$ extending m.*
2. *If t is disabled by a negative pattern $N(Q)$ (i.e. $m' : t \hookrightarrow Q$ is in $Patt^-(t)$) then there is no embedding of Q in $CTrG$ extending m.*

$CTrG$ has no incorrect traces at strict level *if, in addition, in condition 1 it is required that m'' can not be extended to an embedding of any negative pre-condition C_i in $CTrG$.*

Remark. One could try to express condition 2 (for disabling patterns) in a more compact way by just demand-

ing that there is no embedding $m\colon Q^k \to CTrG$ (with Q^k being the main constraint of an N-pattern). However, if Q^k does not contain any trace triple, traces may still be correct in $CTrG$.

Example. Fig. 28 shows a constraint $CTrG$ with one occurrence of the trace triple t_i. The trace is correct with respect to pattern `Class-Table` at the relaxed level because it is contained in one occurrence of the main constraint Q. However, the trace is incorrect at the strict level because such occurrence of the main constraint does not satisfy the pattern's negative pre-condition C_1 (i.e. there is an embedding of C_1 in $CTrG$ extending the embedding of t_i).

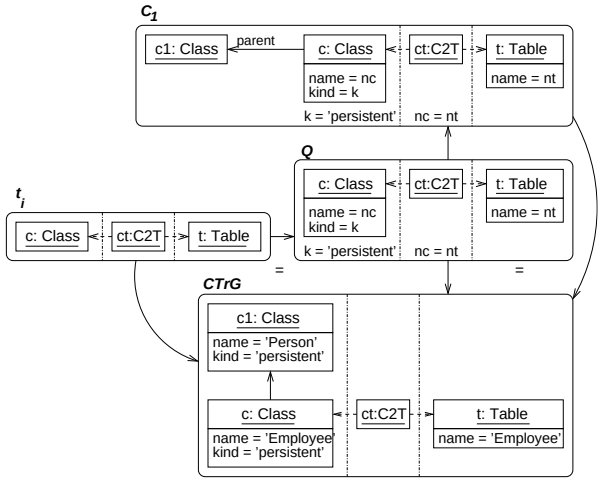

Fig. 28 Detection of incorrect trace at the strict level.

Next we show how to generate operational rules that delete the incorrect traces. In the relaxed case we generate two kinds of rules. The first type is derived from $Patt^+$, and has each $t_i \in Trc$ as LHS and $t_i|_{ST}$ as RHS, so that the trace is deleted. These rules are added each $tr_i \to Q^k \in Patt^+(t_i)$ as NAC, so that the trace is not deleted if it occurs in the main constraint of a P-pattern that demands its presence. The second kind of rules is derived from $Patt^-$, and deletes one trace contained in the occurrence of an N-pattern. In the strict case we generate an additional set of rules that delete a trace if all occurrences of its enabling patterns where it is included violate some negative pre-condition, since these traces cannot be generated by our M2M transformation operational mechanisms.

Definition 14 (Deleting rules for incorrect traces) *Given a specification $SP = \bigwedge_{k\in K} CP^k$, we generate the following rules for deleting incorrect traces at the relaxed level, for each trace $t \in Trc$:*

1. *$del^+(t)$ is the rule with t as LHS, $t|_{ST}$ as RHS (i.e. t without the trace part), and whose NACs are all embeddings $t \hookrightarrow Q$ in $Patt^+(t)$.*
2. *$del^-(t)$ is the set consisting of all rules with Q as LHS, and Q' as RHS, where $N(Q)$ is an N-pattern in SP, $m : t \hookrightarrow Q$ is an embedding of t in Q, and Q' is the triple constraint obtained by deleting from Q the trace part of t at location m.*

At the strict level, we also generate the following rules for each trace $t \in Trc$ and each enabling pattern $CP \in Patt^+(t)$ with $N_{pre}(CP) \neq \emptyset$:

3. *$del^+(t, CP)$ is the rule with t as LHS, $t|_{ST}$ as RHS, and which in addition includes a set PAC of positive application conditions and a set AC of application conditions. More precisely, PAC consists of all embeddings $t \hookrightarrow C_j$, where C_j is a negative pre-condition in CP, and AC consists of all the pairs $(t \hookrightarrow Q', N_{pre}(CP'))$ formed by an embedding of t in Q' and the set of negative pre-conditions of CP', for all enabling patterns $CP' \in Patt^+(t)$ different from CP.*

Remark. The strict rules have two kinds of application conditions that we have not encountered so far. The first one is called PAC (Positive Application Condition) [16] and demands the existence of an additional context C (where $L \hookrightarrow C$) in order to apply the rule. Contrary to NACs, applying a rule requires finding an occurrence of *some* PAC in the set of PACs. In this way, the set PAC in the rule definition requires that the trace to be deleted is included in some negative pre-condition of the enabling pattern C^k. The second kind of Application Condition (AC) is made of an embedding $L \hookrightarrow C$ (the *premise*) and a set of *consequences* C_i embedding C. An AC is satisfied if for any occurrence of its premise we find an occurrence of some of its consequences. Actually, a NAC is just an AC without consequences, and hence finding the premise disables the rule application. The set AC in the definition checks that any trace embedded in the occurrence of other enabling pattern CP', is embedded in a negative pre-condition occurrence of such pattern as well, so that the trace can be safely deleted.

Example. Fig. 29 shows the first type of relaxed deleting rule generated for the trace triple relating classes and tables. Thus, a `C2T` trace between a table and a class will be deleted unless the class is persistent and has same name as the matched table (`NAC1`), and it has no parent connected to the table (`NAC2`). The two NACs come from the enabling patterns of the trace triple shown in Fig. 26.

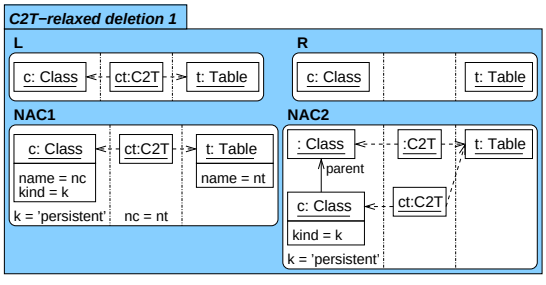

Fig. 29 First type of relaxed deleting rule for `C2T` traces.

Fig. 30 shows the strict deleting rule derived for the same trace triple and the pattern `Class-Table`. The rule is generated because the pattern contains the trace triple and has a negative pre-condition. The rule's PAC comes from the pattern negative pre-condition and allows executing the rule only if the class has a parent. The rule also contains an AC derived from the other enabling pattern of the trace triple, which is `Parent Class-Table`. As `Parent Class-Table` has no negative pre-conditions, the AC becomes a NAC. This avoids deleting the trace if the class is persistent and has a parent traced to the same table.

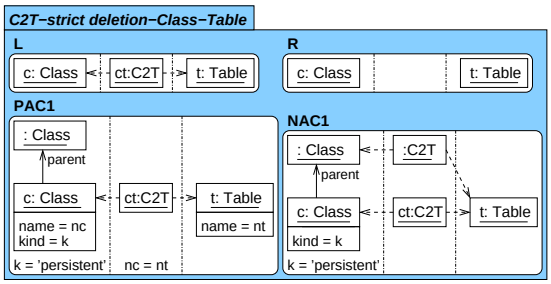

Fig. 30 Strict deleting rule for `C2T` traces and pattern `Class-Table`.

Fig. 31 shows one scenario where initially the first type of relaxed rule and then the strict rule are applied. The top-most model is the starting point, where a class and its child are related to the same table. Applying the first rule deletes the trace between the parent class and the table as they have different names. Note that in the initial scenario the strict rule is not applicable because, even though class $c2$ does have a parent, this is connected to the same table as $c2$. In the second step, the relaxed rule is no longer applicable, and it is possible to apply the strict rule. This deletes the trace between the child class and the table as the child class has a parent, which is not connected to the table. Even if both the intermediate and the final models are correctly traced, we prefer the last one as the intermediate one is not generated by our operational mechanism (M2M, matching or traceability).

The generated relating (matching or traceability) and deleting rules are incremental. Thus, given a source and a target models connected through an arbitrary trace model, we apply the rules as long as possible: the deleting rules will delete the incorrect traces, and the creating rules will reestablish trace correctness. We believe the notions presented in this section are a first step towards incremental synchronization, as we are able to identify the places where two related models are not correctly traced. Here we took the solution of deleting the trace, but other operational mechanisms could modify the source or the target to make the models consistent again. Moreover, the traceability rules generate all possible traceability solutions, which can be the basis to reach all possible models synchronized with respect to the specification.

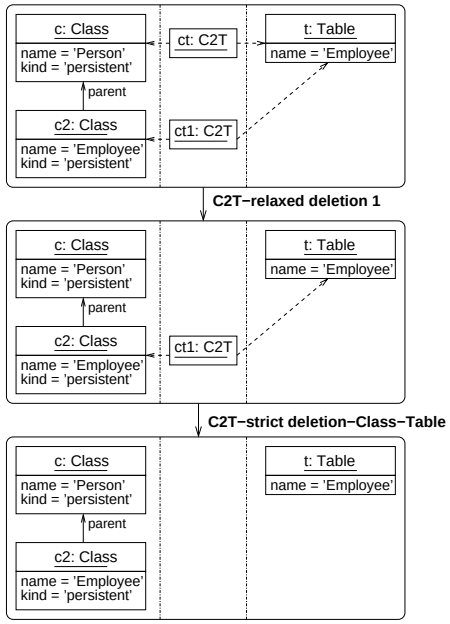

Fig. 31 Applying a relaxed and then a strict rules.

8 Example

In this section we illustrate our approach with an inter-model specification between relational database schemas (RDBMS) and XML documents. The meta-model triple is shown in Fig. 32. Schemas contain books and subjects. A book has zero or more subjects, and those books with the same subject description are related to the same object *Subject*. On the contrary, the XML meta-model allows nested relationships, and even if two books have the same subject description, they are assigned two different objects *Subject*.

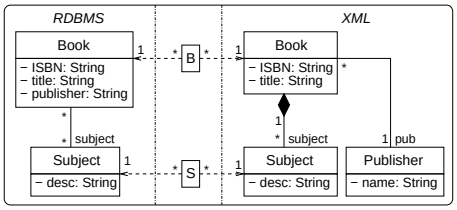

Fig. 32 Meta-model triple for RDBMS and XML.

Fig. 33 shows the inter-model specification made of four patterns. The P-pattern `Book` states how books in both meta-models must relate, and adds an ".ed" suffix to the publishers in the XML model. P-pattern `Subject` maps subjects in both models. Note that we need these two patterns as it is possible to have books with zero or more subjects. Should books always have exactly one subject, then only one pattern would have been enough.

In addition, as the RDBMS format does not allow two subjects with the same description, we forbid such situation by defining the N-pattern `N(NotDupRDBMSSubject)`. Similarly, N-pattern `N(NotDupXMLPublisher)` forbids repeating publishers in XML.

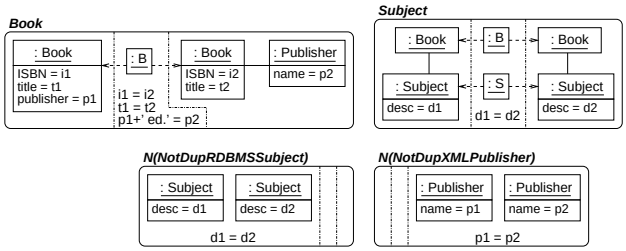

Fig. 33 Initial model-to-model transformation specification.

The specification is automatically enriched with additional N-patterns derived from the maximum cardinality constraints in the meta-model triple. In particular, the N-patterns shown in Fig. 34 are generated and added to the specification. The left one forbids a subject to belong to two books, as the association end with cardinality *1* between subjects and books indicates, whereas the right one forbids a book to have two publishers.

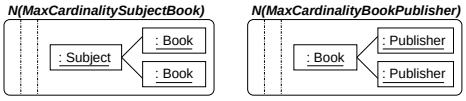

Fig. 34 Additional N-patterns from the meta-model.

In this example we do not use parameterization but use the heuristics instead. The heuristics generate the patterns shown in Figs. 35 and 36. In particular, the heuristic 1 generates pattern `Subject.h1` from pattern `Subject` by adding the elements with unconstrained attributes as parameters. The new pattern replaces the old one and ensures that, when a subject is translated, the book associated to it has been translated first.

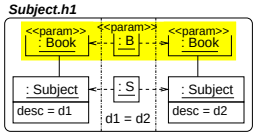

Fig. 35 New pattern generated by heuristic 1.

The heuristic 2 is applied to patterns `Subject.h1` and `Book` and produces the patterns `Subject.h1.h2` and `Book.h2` shown in Fig. 36. The first one reuses RDBMS subjects so that they are not duplicated in backward transformations. The second reuses publishers avoiding its duplication in forward transformations. Note that the heuristic is not applicable to the N-patterns of Fig. 34 because, although they forbid repetition of books and publishers, no P-pattern demands a book in the main constraint given a subject in its parameter, and similarly, no pattern demands a publisher in the main constraint given a book in its parameter.

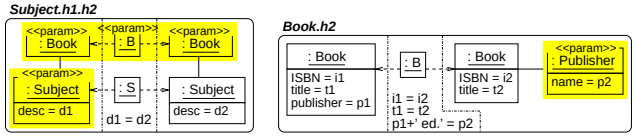

Fig. 36 New patterns generated by heuristic 2.

8.1 Forward Transformation Scenario

In the forward M2M transformation scenario, we use the final specification, made of P-patterns `Book`, `Subject.h1`, `Subject.h1.h2` and `Book.h2`, as well as all N-patterns, to generate the operational forward rules shown in Fig. 37. Rule `Book` contains a termination NAC equal to R, and a negative post-condition (generated from `N(NotDupXMLPublisher)`) avoiding two publishers with same name. Rule `Book.h2` creates books that reuse publishers once they have been created. Its negative post-condition is generated from `N(MaxCardinalityBookPublisher)`. Finally, patterns `Subject.h1` and `Subject.h1.h2` produce equivalent rules with two termination NACs. Note that we do not need to perform algebraic manipulation of expressions for rule synthesis, as the LHS and the RHS in the rules contain constraint triple graphs. Recall that the attributes not used in formulae are omitted, like e.g. in the LHS of rule `Book`.

The forward operational rules generated in this example are terminating, confluent[4], correct and complete even using heuristics. However, in general, the rules we generate cannot guarantee confluence if we do not have a means to prefer one resulting model or another.

As an example, Fig. 38 presents all possible execution flows to transform the RDBMS model shown to the left. Starting from this model we can apply rule `Book` to two different locations, one for each book, obtaining two different models. Starting from them, in both cases we can apply either rule `Subject.h1` or rule `Book.h2`, leading to different models (which we not show for space limitations but represent them as squares instead), to which again we can apply different rules yielding different models. In any case, as we said before, this transformation is confluent and we obtain the final model shown to the right independently of the followed execution flow (up to renaming of variable names).

[4] The analysis of confluence was done using the AGG tool, see `http://tfs.cs.tu-berlin.de/agg`.

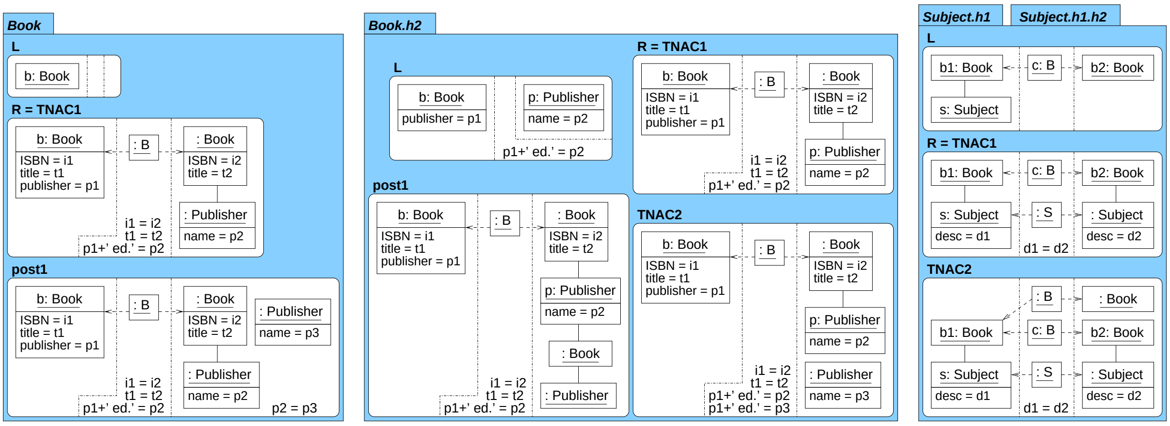

Fig. 37 Generated forward rules.

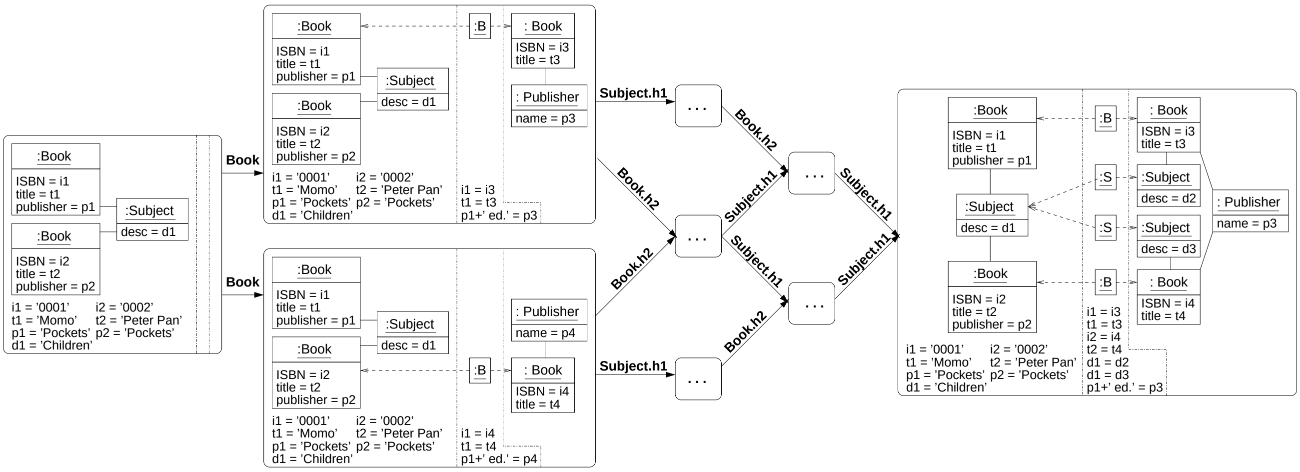

Fig. 38 Model-to-model transformation.

8.2 Traceability Scenario

The set of patterns obtained after using heuristics generates two traceability rules. The upper row of Fig. 39 shows one of them, which creates the traces between books and is derived from patterns `Book` and `Book.h2`. The second rule (not shown) creates traces between subjects and is derived from patterns `Subject.h1` and `Subject.h1.h2`.

The figure shows below the relaxed deleting rule that deletes incorrect traces between subjects. The two enabling patterns of the trace (`Subject.h1` and `Subject.h1.h2`) generate the same and unique NAC. As these patterns have no negative pre-conditions, there is no need to derive strict deletion rules from them. Another deleting rule (not shown) is generated to delete incorrect traces between books.

Now, assume that the resulting models in Fig. 38 are changed by the user: he attaches a new subject *Teen* to the RDBMS book *Momo*, and modifies the subject of the XML book *Momo* from *Children* to *Teen*. The upper model in Fig. 40 depicts this situation, where it can be seen how the trace model is no longer correct. Hence, we apply the generated tracing and deleting rules to obtain trace consistency. The deleting rules delete the incorrect trace between subject *Children* to the left and

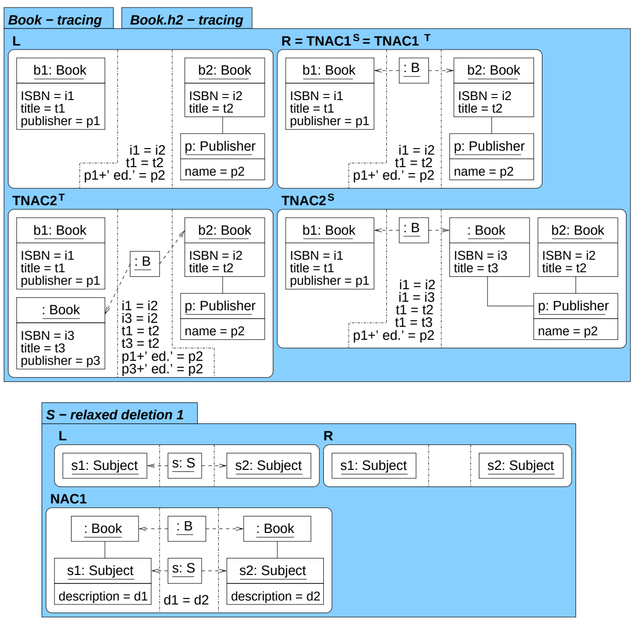

Fig. 39 Some traceability and relaxed deleting rules.

subject *Teen* to the right as they have different descriptions. Then, the traceability rules create a trace between the subjects *Teen* to the left and to the right. The result is shown in the same figure below, where the trace model is again correct.

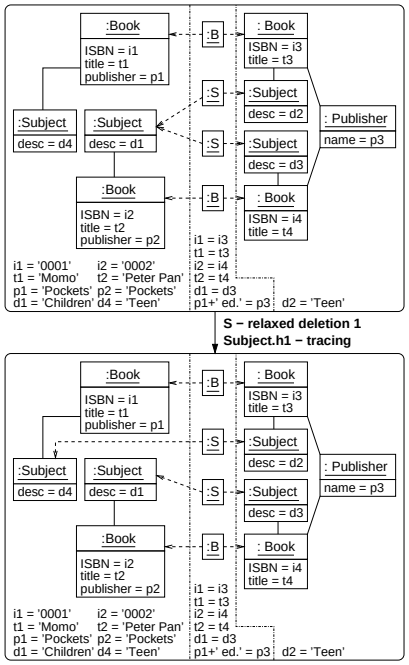

Fig. 40 Trace model repair.

8.3 Matching Scenario

As a scenario for matching, consider one RDBMS and one XML model built by different companies. We would like to match these two models identifying similar books and subjects, with the purpose of merging the two models into an integrated one. The task of matching can be performed again using the same specification.

In scenarios like this one, a common necessity is approximate matching of string names for certain features, e.g. match equivalent descriptions of subjects such as *Children* and *Kids*. We can use the loose specification capabilities of our approach to provide an "approximate pattern". For instance, in Fig. 41 there is a more flexible version of pattern `Subject`, which allows matching subjects with equal description, but also with descriptions that conceptually refer to the same subject matter, as specified in its attribute condition. Hence, the matching rule generated from this pattern will be able to match, e.g., two subjects with descriptions *Teen* and *Youth*.

This new pattern is also usable for M2M transformation, and results in a loose specification where a source model admits several target models as solution. That is, the generated forward rule will make two solutions available for the description of the target subject when the source one is *Kids*.

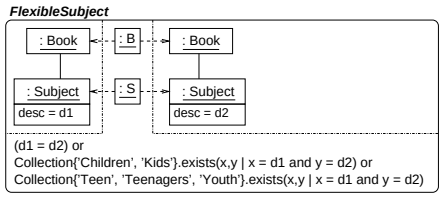

Fig. 41 Additional pattern for model matching, considering a complex attribute condition.

9 Conclusions and Future Work

In this paper we have presented our pattern-based approach, which is able to handle three inter-modelling scenarios: forward and backward batch M2M transformation of models, model matching and model traceability. Our inter-modelling language allows expressing relations between models in a declarative way, using both structural object patterns and declarative attribute conditions. The advantage of our approach is that it provides a formal, high-level language to express inter-model relations, which can be used to solve several scenarios. Our language is concise, as its heuristics allow omitting the parameters in the relations. Moreover, at the operational level, we have proposed a new way of triple graph rewriting based on constraints. This idea, which can be used in other transformation approaches, avoids manipulation of attribute conditions, one of the main difficulties of relational approaches.

In the three considered inter-modelling scenarios we have provided: (i) a formal notion of satisfaction of specifications by models, and (ii) operational mechanisms able to re-establish consistency in the scenarios. The notion of satisfaction is given with a high-level, algebraic semantics that is independent of the operational mechanism. This contrasts with most approaches, where the specification and the operational mechanisms are highly tied, e.g. based on parsing with rules as in TGGs, or in terms of QVT-Core as QVT-Relations.

We believe our language is appropriate to serve as a unifying, formal and visual framework for inter-modelling. This is so because a unique specification can be used in a flexible way to solve different scenarios. There is a trade-off between operational and relational languages for inter-modelling, though. Operational languages include low-level, operational primitives that make them more expressive, at the cost of being close to a general purpose programming language. In contrast, relational approaches are restricted in expressivity, but are higher level as they do provide a model of the allowed relations, rather than a program to perform each considered scenario. Instead, such programs are derived from the specification, depending on the scenario to be solved. Relational approaches thus are more suitable for inter-modelling, as they produce high-level models.

9.1 Tool Support

We are currently implementing tool support for the presented framework. In particular, we have developed an Eclipse tool to build pattern specifications using a textual syntax. The tool is available at `http://astreo.ii.uam.es/~eguerra/tools/pamomo/main.htm`, together with the example of Section 8. Patterns can be compiled into operational rules for certain scenarios. The generated rules are implemented with EOL [31], which is an extension of OCL. We chose this target language because OCL is an OMG standard and can be integrated in transformation languages of widespread use, such as ATL, ETL or QVT.

For the time being, the tool is able to check the satisfaction of specifications by models in all scenarios (M2M transformation, matching and traceability). It is also possible to populate the traces of two models in the model matching and traceability scenarios, as well as to delete incorrect traces.

In the short term, we plan to extend the tool to handle forward/backward transformations. However this is the most complex scenario since the resolution of attribute values requires the combined use of transformation rules and constraint solvers, possibly at run-time to enhance efficiency.

9.2 Lines of Future Work

The work presented here opens many lines for further research. For example, we are interested in integrating our pattern specifications with approaches enabling richer trace models for model traceability [14]. We are also considering other inter-modelling scenarios like model synchronization and model merging. Note that for the latter we will need a third model, and hence to extend our formalization. We are also thinking of compiling our patterns into other operational languages, like Coloured Petri nets as in [12], as well as devising more analysis methods like those in [39]. Finally, it is worth investigating the relations of our pattern language with that of QVT-R. Even though in [24] we adapted the semantics of our patterns for the QVT-R checkonly scenario, further work to handle other inter-modelling scenarios with QVT-R is needed.

Acknowledgements. This work has been sponsored by the Spanish Ministry of Science and Innovation with projects METEORIC (TIN2008-02081) and FORMALISM (TIN2007-66523), and by the R&D program of the Community of Madrid (S2009/TIC-1650, project "e-Madrid"). This work was done during the research stays of the first two authors at the University of York, with financial support from the Spanish Ministry of Science and Innovation (grant refs. JC2009-00015, PR2009-0019 and PR2008-0185). We would like to thank the referees for their useful and detailed comments, as well as the SOSYM group at the University of York for the discussions on this topic.

References

1. D. H. Akehurst and S. Kent. A relational approach to defining transformations in a metamodel. In *UML'02*, volume 2460 of *LNCS*, pages 243–258. Springer, 2002.
2. M. Aleksy, T. Hildenbrand, C. Obergfell, and M. Schwind. A pragmatic approach to traceability in model-driven development. In *PRIMIUM'08*, volume 328 of *CEUR*. CEUR-WS.org, 2008.
3. ATL. `http://www.sciences.univ-nantes.fr/lina/atl/`.
4. J. Bézivin, F. Büttner, M. Gogolla, F. Jouault, I. Kurtev, and A. Lindow. Model transformations? Transformation models! In *MoDELS'06*, volume 4199 of *LNCS*, pages 440–453. Springer, 2006.
5. A. Boronat, J. A. Carsí, and I. Ramos. Exogenous model merging by means of model management operators. In *SeTra'06*, volume 3. ECEASST, 2006.
6. P. Braun and F. Marschall. Transforming object oriented models with BOTL. *ENTCS*, 72(3):103–117, 2003.
7. G. Brunet, M. Chechik, S. Easterbrook, S. Nejati, N. Niu, and M. Sabetzadeh. A manifesto for model merging. In *GaMMa'06*, pages 5–12. ACM Press, 2006.
8. J. Cabot, R. Clariso, E. Guerra, and J. de Lara. Verification and validation of declarative model-to-model transformations through invariants. *JSS*, 83:283–302, 2010.
9. J. Cleland-Huang, J. H. Hayes, and J. M. Domel. Model-based traceability. In *TEFSE'09*, 2009.
10. J. de Lara, R. Bardohl, H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. Attributed graph transformation with node type inheritance. *TCS*, 376(3):139–163, 2007.
11. J. de Lara and E. Guerra. Pattern-based model-to-model transformation. In *ICGT'08*, volume 5214 of *LNCS*, pages 426–441. Springer, 2008.
12. J. de Lara and E. Guerra. Formal support for QVT-relations with coloured petri nets. In *MoDELS'09*, volume 5795 of *LNCS*, pages 256–270. Springer, 2009.
13. M. Dehayni and L. Féraud. An approach of model transformation based on attribute grammars. In *OOIS*, volume 2817 of *LNCS*, pages 412–424. Springer, 2003.
14. N. Drivalos, D. Kolovos, R. Paige, and K. Fernandes. Engineering a DSL for software traceability. In *SLE'08*, volume 5452 of *LNCS*, pages 151–167. Springer, 2008.
15. H. Ehrig, K. Ehrig, C. Ermel, F. Hermann, and G. Taentzer. Information preserving bidirectional model transformations. In *FASE'07*, volume 4422 of *LNCS*, pages 72–86. Springer, 2007.
16. H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of algebraic graph transformation*. Springer-Verlag, 2006.
17. H. Ehrig, C. Ermel, F. Hermann, and U. Prange. On-the-fly construction, correctness and completeness of model transformations based on triple graph grammars. In *MoDELS'09*, volume 5795 of *LNCS*, pages 241–255. Springer, 2009.
18. H. Ehrig, F. Hermann, and C. Sartorius. Completeness and correctness of model transformations based on triple graph grammars with negative application conditions. In *GT-VMT'09*, volume 18. ECEASST, 2009.

19. A. Espinoza, P. P. Alarcón, and J. Garbajosa. Analyzing and systematizing current traceability schemas. In *SEW'06*, pages 21–32. IEEE CS, 2006.
20. J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.*, 29(3), 2007.
21. H. Giese and R. Wagner. From model transformation to incremental bidirectional model synchronization. *SoSyM*, 8(1):21–43, 2009.
22. M. Goedicke, B. Enders, T. Meyer, and G. Taentzer. Towards integration of multiple perspectives by distributed graph transformation. In *AGTIVE'99*, volume 1779 of *LNCS*, pages 369–377. Springer, 1999.
23. E. Guerra and J. de Lara. Event-driven grammars: Relating abstract and concrete levels of visual languages. *SoSyM*, 6(3):317–347, 2007.
24. E. Guerra, J. de Lara, D. S. Kolovos, and R. F. Paige. A visual specification language for model-to-model transformations. In *Proc. IEEE VL/HCC'10*, pages 119–126, 2010.
25. E. Guerra, J. de Lara, and F. Orejas. Pattern-based model-to-model transformation: Handling attribute conditions. In *ICMT'09*, volume 5563 of *LNCS*, pages 83–99. Springer, 2009.
26. E. Guerra, J. de Lara, and F. Orejas. Controlling reuse in pattern-based model-to-model transformations. In *Festschrift for Manfred Nagl (65th birthday)*, volume 5765 of *LNCS*, pages 178–204. Springer, 2010.
27. I. Ivkovic and K. Kontogiannis. Tracing evolution changes of software artifacts through model synchronization. In *ICSM'04*, pages 252–261. IEEE CS, 2004.
28. KMF. `http://www.cs.kent.ac.uk/projects/kmf/`.
29. D. Kolovos, D. Di Ruscio, A. Pierantonio, and R. Paige. Different models for model matching: An analysis of approaches to support model differencing. In *CVSM'09*, pages 1–6, 2009.
30. D. S. Kolovos. Establishing correspondences between models with the Epsilon Comparison Language. In *ECMDA-FA'09*, volume 5562 of *LNCS*, pages 146–157. Springer, 2009.
31. D. S. Kolovos, R. F. Paige, and F. Polack. The Epsilon Object Language (EOL). In *ECMDA-FA'06*, volume 4066 of *LNCS*, pages 128–142. Springer, 2006.
32. D. S. Kolovos, R. F. Paige, and F. Polack. The Epsilon Transformation Language. In *ICMT'08*, volume 5063 of *LNCS*, pages 46–60. Springer, 2008.
33. A. Königs and A. Schürr. Tool integration with triple graph grammars - a survey. *ENTCS*, 148(1):113–150, 2006.
34. M. Lawley and J. Steel. Practical declarative model transformation with Tefkat. In *MoDELS Satellite Events*, volume 3844 of *LNCS*, pages 139–150. Springer, 2005.
35. P. Mäder, O. Gotel, and I. Philippow. Rule-based maintenance of post-requirements traceability relations. In *RE'08*, pages 23–32. IEEE CS, 2008.
36. S.-C. Mu, Z. Hu, and M. Takeichi. Bidirectionalizing tree transformation languages: A case study. *JSSST Computer Software*, 23(2):129–141, 2006.
37. S. Nejati, M. Sabetzadeh, M. Chechik, S. Easterbrook, and P. Zave. Matching and merging of statecharts specifications. In *ICSE'07*, pages 54–64. IEEE CS, 2007.
38. F. Orejas, E. Guerra, J. de Lara, and H. Ehrig. Correctness, completeness and termination of pattern-based model-to-model transformation. In *CALCO'09*, volume 5728 of *LNCS*, pages 383–397. Springer, 2009.
39. F. Orejas and M. Wirsing. On the specification and verification of model transformations. In *Semantics and Algebraic Specification*, volume 5700 of *LNCS*, pages 140–161, 2009.
40. QVT. `http://www.omg.org/docs/ptc/05-11-01.pdf`.
41. M. Rebout, L. Féraud, and S. Soloviev. A unified categorical approach for attributed graph rewriting. In *CSR'08*, volume 5010 of *LNCS*, pages 398–409. Springer, 2008.
42. J. Sánchez, J. García, and M. Menárguez. RubyTL: A practical, extensible transformation language. In *ECMDA-FA'06*, volume 4066 of *LNCS*, pages 158–172. Springer, 2006.
43. A. Schürr. Specification of graph translators with triple graph grammars. In *WG'94*, volume 903 of *LNCS*, pages 151–163. Springer, 1994.
44. A. Schürr and F. Klar. 15 years of triple graph grammars. In *ICGT'08*, volume 5214 of *LNCS*, pages 411–425. Springer, 2008.
45. SmartQVT. `http://smartqvt.elibel.tm.fr/`.
46. P. Stevens. Bidirectional model transformations in QVT: Semantic issues and open questions. In *MoDELS'07*, volume 4735 of *LNCS*, pages 1–15. Springer, 2007.
47. J. Winkelmann, G. Taentzer, K. Ehrig, and J. M. Küster. Translation of restricted OCL constraints into graph constraints for generating meta model instances by graph grammars. *ENTCS*, 211:159–170, 2008.
48. Z. Xing and E. Stroulia. UMLDiff: an algorithm for object oriented design differencing. In *ASE'05*, pages 54–65. ACM, 2005.

A Appendix

This appendix provides the proofs of the claims and propositions in the paper. The details of the construction of pushouts in the category of constraint triple graphs can be found in the appendix of [26].

Proof of Proposition 1.

Proof We have to prove that $SEM(SP_1 \wedge SP_2) = SEM(SP_1) \cap SEM(SP_2)$, which easily follows as $SEM(SP_1 \wedge SP_2) = \{G | G \models SP_1 \wedge G \models SP_2\}$, and $SEM(SP_1) \cap SEM(SP_2) = \{G | G \models SP_1\} \cap \{G' | G' \models SP_2\} = \{G | G \models SP_1 \wedge G \models SP_2\} = SEM(SP_1 \wedge SP_2)$.

Proof of Proposition 2.

Proof The first part of the proposition states that a model of the specification in the transformation scenario is also a model of the specification in the traceability scenario. In other words, if two graphs are synchronized, then they are correctly traced: $CTrG \models SP \Rightarrow CTrG \models^T SP$. Synchronization demands each pattern $CP \in SP$ that in each occurrence when it is source or target enabled it is actually satisfied. That is, $\forall m^S \colon P_S \to CTrG$ s.t. $CTrG \vdash_{m^S,F} CP$

then $CTrG \models_{m^S,F} CP$, and $\forall m^T \colon P_T \to CTrG$ s.t. $CTrG \vdash_{m^T,B} CP$ then $CTrG \models_{m^T,B} CP$ (where P_S and P_T are CP's forward and backward pre-conditions). But in every occurrence of the trace pre-condition $m^{ST} \colon P_{ST} \to CTrG$ that is trace-enabled $CTrG \vdash_{m^{ST},T} CP$, there are restrictions $m^S \colon P_S \to CTrG$ and $m^T \colon P_T \to CTrG$ that are forward and backward enabled respectively. Trace satisfaction demands that in each occurrence m^{ST} either m^S or m^T are satisfied (it is traced by Q). However, by the definition of synchronization, both will be satisfied. The implication does not hold in the other direction though, as we allow untraced elements in the source (resp. target), if there are not enough untraced elements in the target (resp. source).

The second part of the proposition states that a model of the specification in the matching scenario is also a model of the specification in the traceability scenario. In other words, if two graphs are matched, then they are traced: $CTrG \models^M SP \Rightarrow CTrG \models^T SP$. This easily follows as the matching satisfaction condition demands that each trace pre-condition $m^{ST} \colon P_{ST} \to CTrG$ that is trace-enabled $CTrG \vdash_{m^{ST},T} CP$ has to be satisfied (hence both m^S and m^T are satisfied in an occurrence of Q that embeds m^{ST}). The traceability satisfaction condition is weaker, as it states that in each trace pre-condition $m^{ST} \colon P_{ST} \to CTrG$ that is trace-enabled $CTrG \vdash_{m^{ST},T} CP$, either m^S or m^T are satisfied. Therefore, if a constraint $CTrG$ is matched, it is traced.