On the specification and verification of model transformations

Fernando Orejas[1] and Martin Wirsing[2]

[1] Universitat Politècnica de Catalunya, Barcelona (Spain), orejas@lsi.upc.edu
[2] Ludwig-Maximilians Universität, Munich (Germany), wirsing@pst.ifi.lmu.de

Abstract. Model transformation is one of the key notions in the model-driven engineering approach to software development. Most work in this area concentrates on designing methods and tools for defining or implementing transformations, on defining interesting specific classes of transformations, or on proving properties about given transformations, like confluence or termination. However little attention has been paid to the verification of transformations. In this sense, the aim of this work is, on one hand, to clarify what means to verify a model transformation and, on the other, to propose a specific approach for proving the correctness of transformations. More precisely, we use some general patterns to describe both the transformation and the properties that we may want to verify. Then, we provide a method for proving the correctness of a given transformation.

1 Introduction

Model transformation plays a central role in the model-driven engineering approach to software development. In this context, a model is an abstract description of an artifact, and model transformations are used to build new models out of existing ones. There are many kinds of model transformations that are considered interesting [18]. For instance, transformations going from more abstract to more concrete models may be part of the refinement process for yielding the final implementation of a system. Conversely, a transformation yielding more abstract models out of more concrete models may be used used when doing reverse engineering, for example in connection with legacy systems. There are also transformations where the source and target models share the same level of abstraction. For example, this is the case of refactorings, where we change the structure of a system perhaps for improving its performance. From a different point of view, model transformations may be endogenous or exogenous. In endogenous transformations the source and the target model belong to the same class of models. This means that the transformation is defined within the same metamodel. This is the case, for example, of refactorings. Conversely, in an exogenous transformation the source and the target models may be of a very different kind. Exogenous transformations may occur for instance when the source and the target model describe a given system with a different level of abstraction.

Most work in the area of model transformation concentrates on designing methods and tools for defining or implementing transformations, on defining interesting specific classes of transformations, or on proving properties about given transformations, like

confluence or termination. However little attention has been paid to the problem of ensuring that a given transformation is correct. Where *correct* means that the source and the target models are in some sense a description of the same artifact. Possibly, one of the reasons why there is not much work on this topic is the difficulty, in the case of exogenous transformations, of having to deal with different kinds of models. Actually, one of the few approaches [27] that we know on the verification of model transformations concentrates on the case of endogenous transformations and, in particular, in the case of refactorings, although their techniques would be applicable to other kinds of endogenous transformations. The difficulty of dealing, on a uniform setting, with the verification of exogenous model transformations is covered in [2] by working at the level of *institutions* [10]. More precisely, in that paper the authors consider that in an exogenous transformation the source and the target models live on different institutions. Then, the correctness of a transformation is stated in terms of the existence of an intermediate institution that relates the source and target institutions through a span of an institution morphism and an institution comorphism. The idea is that the models in this intermediate institution include the common aspects of the source and the target models. The approach, which in some sense is conceptually clarifying and has influenced the present work, has two main problems from our point of view. On one hand, working at this multi-institutional level makes difficult to develop specific methods and tools for the verification of transformations. A solution to this problem, which is in a way implicit in [2] and which has been pursued further in [1], consists in the representation of all metamodels in a unifying framework, in this case Rewriting Logic. Unfortunately, as said above, this possible solution is only implicit in [2], while the work presented in [1] is, in a way, not directly related. Although in [1] they approach the verification problem, it is a different verification problem. More precisely, they are not concerned on whether a given transformation is correct in the above sense, but on how, by using their approach in connection with the Maude system, one can verify the resulting model, in the sense, for example, of showing that it is deadlock-free. The second problem of [2] is that they assume that it is possible to verify a model transformation by itself, without any additional information, by showing that the source and the target models are, in some sense, semantically equivalent. We believe that this is very reasonable for certain kinds of transformations, for example refactorings, but not in general. From our point of view, the correctness of a model transformation depends on what we are interested to observe in the source and target models. For instance, suppose that we have a model describing a system that processes some input coming from some components. Moreover, suppose that this model describes that this input is processed according to the order of arrival. Now, suppose that we apply a transformation to this model and, in the resulting target model, the input is not processed according to the order of arrival. This transformation would be incorrect if the order in which the input is processed is important, but it may be correct otherwise.

The work that we present is also influenced by the previous work of the first author on the use of graph constraints for modelling of software systems [23, 22] and, especially, for the specification of model transformation [24, 12]. More precisely, in [6], de Lara and Guerra introduced a visual declarative method for specifying bidirectional model transformation based on the use of what they called *triple patterns*. This method

was itself based on the use of triple graphs [28] to describe model transformations and on the use of graph constraints for software modelling. In [24] we showed how a specification specification of this kind could be translated into a finitely terminating graph transformation system, where the valid transformations were terminal graphs of the transformation system, and in [12] we extended the specification approach to deal with attributes over predefined data types, using a similar technique to [22]. In the current paper, we follow a similar approach to [12] for the specification of model transformations, but instead of considering that models are characterized by graphs, we generalize the approach dealing instead with algebras. The main reason is the limited expressive power of graph constraints and triple patterns. For example, if class diagrams are represented as typed graphs and the subclass relationship is represented by edges, then using graph constraints or triple patterns it is impossible to express that a certain class is a non-direct subclass of another class. This is not a problem when graphs are replaced by algebras. On the other hand, dealing with models as algebras would allow us to use a tool as MAUDE in the implementation of our results.

Nevertheless, the main contribution of this paper is related to the verification of model transformations, rather than with their specification. More precisely, after some conceptual discussion, we present a specific method, based also on the use of patterns, to specify the properties that would specify the correctness of a given transformation and we give the basis for a method for checking that a transformation is correct. To describe and motivate our concepts and results we use the standard example [26] of the transformation of class diagrams into relational schemas.

The paper is structured as follows. In the following section we present some basic algebraic notation and terminology at the same time that we discuss its use in the context of the model-driven engineering framework. Then, in Section 3 we introduce the notion of algebraic constraint as the translation of the notion of graph constraint when dealing with algebras instead of graphs. In this section we also introduce the category of triple algebras, following the ideas of the triple graph approach, and we show how we can use (triple) patterns to describe properties of a model transformation. Next, in Section 4 we generalize the specification method presented in [6, 12] to the algebraic case. In the 5th section we address the problem of verifying model transformations. We discuss what it means to verify a model transformation and we propose a specific approach including, as mentioned above, a method for checking that a transformation is correct. Finally, in Section 6, we draw some conclusions and discuss further work.

2 Metamodels, Models and Instances

In the context of the Model-Driven Engineering approach (MDE), a model is a description of a system. This means that a model is what in different contexts is called a specification. As pointed out in [2] this may cause some confusion in terminology. Then, following [2], models in the sense of the MDE will often be called *software engineering models*. A software engineering model may describe the structure of a system. This is the case, for instance, of a class diagram. These models are called *structural models*. Otherwise, they may describe the behaviour of a system, like Petri nets or state

diagrams. In that case they are called *behavioral models*. In this paper we will essentially concentrate on structural models.

A class of software engineering models is defined by means of a *metamodel*. In the MDE a metamodel essentially describes the structure of models, but not its semantics, which is left implicit. This is typically done by means of a class diagram. But this is not necessarily the only kind of metamodel specification. In[3, 1] metamodels are defined by means of algebraic specifications using Maude [5]. In that approach, a signature plus some axioms play the same role as class diagrams to describe the structure of a model. Actually, they provide in this way a specification of class diagrams. In that context, the initial algebra of a (metamodel) specification can be seen as the class of all the software engineering models associated to the given metamodel. This means that every ground term over the given signature may be considered to denote a software engineering model. Metamodel definition is not relevant for the work presented in this paper. Therefore, we will not assume any concrete form of metamodel specification.

As said above, a software engineering model is a specification of a system. In the case of structural models, this specification describes the different kinds of parts of the given system. For instance, in a class diagram we specify the classes of objects that our system may include, and the relations between these classes. This means that, again, from an algebraic point of view, we can see a structural model as a signature, perhaps together with some axioms (for instance, an axiom may state that inheritance is transitive). In many cases, the signatures associated to a given kind of model may be just a standard algebraic signature including just some sorts, operations and, perhaps, relations. However, in some other cases, we may consider that the signatures associated to a given kind of model may be in some sense different, for instance they may need to have a richer type structure, like order-sorted signatures [11]. Anyhow, to be concrete and to provide a better intuition, we will assume that the signatures associated to the models considered are standard algebraic signatures. We think that this is not a real limitation. On one hand, using standard signatures, but having a semantics which is not the standard one, we may, in a way, code other kinds of models. For instance, in the approaches presented in [4, 19–21] standard algebras are used, in some sense, to code algebras with a richer type structure. On the other hand, all our constructions and results are based on using some general categorical constructions and properties. This means that, if we replace the specific categories used in this paper by other categories which enjoy the same kind of constructions and properties, then the same results would still apply.

Therefore, we will consider that software engineering models consist of a *signature* $\Sigma = (S, \Omega, \Pi)$ and, perhaps some Σ-axioms Ax, where S is the set of sorts of Σ, $Sorts(\Sigma)$, Ω is the family of operators of Σ, $Ops(\Sigma)$, and Π is the family of relation symbols of Σ, $Rel(\Sigma)$. Then, a *signature morphism* $f : \Sigma_1 \to \Sigma_2$ consists of three mappings, $f_{Sorts} : Sorts(\Sigma_1) \to Sorts(\Sigma_2)$, $f_{Ops} : Ops(\Sigma_1) \to Ops(\Sigma_2)$, and $f_{Rel} : Rel(\Sigma_1) \to Rel(\Sigma_2)$, such that if the arity of $\sigma_1 \in Ops(\Sigma_1)$ is $s_1 \times \cdots \times s_n \to s$ (resp. if the arity of $\rho_1 \in Rel(\Sigma_1)$ is $s_1 \times \cdots \times s_n$) then the arity of $f(\sigma)$ is $f_{Sorts}(s_1) \times \cdots \times f_{Sorts}(s_n) \to f_{Sorts}(s)$ (resp. the arity of $f(\rho)$ is $f_{Sorts}(s_1) \times \cdots \times f_{Sorts}(s_n)$). Signatures and signature morphisms form the category **Sig**.

For instance, a class diagram may be seen as a signature, similar to a graph signature, having one sort for all classes, one sort for associations and one sort for the inheritance relations together with four relation symbols $SourceAssoc: Associations \times Classes$, $Target: Assoc: Associations \times Classes$, $SourceInh: Inheritance \times Classes$ and $TargetInh: Inheritance \times Classes$. In addition, we may consider an axiom stating that inheritance is transitive. An alternative specification would consist in representing class diagrams by signatures where we have a sort for each class in the diagram, where associations and inheritance relations are represented as relation symbols in the signature, and where attributes and methods are represented as operators. This is essentially how class diagrams are represented in [3, 1] . Nevertheless, in this paper we will not define any preferred signature for class diagrams. Instead, in the examples we will directly present them in the standard graphical representation.

The possible states of a system defined by a given software engineering model (Σ, Ax) are often called the *instances* of the model. Using the standard terminology from mathematical logic, instances would be models of (Σ, Ax), i.e. Σ-algebras that satisfy the axioms in Ax.

To be more precise, a Σ-algebra A, as usual, consists of an S-indexed family of carriers, $\{A_s\}_{s\in S}$, an interpretation $\sigma_A: A_{s_1} \times \cdots \times A_{s_n} \to A_s$ for each operator $\sigma: s_1 \times \cdots \times s_n \to s$ in Ω, and an interpretation $\rho_A: A_{s_1} \times \cdots \times A_{s_n}$ for each relation symbol $\rho: s_1 \times \cdots \times s_n$ in Π. A Σ-homomorphism $h: A \to B$ is an S-indexed family of mappings, $\{f_s: A_s \to B_s\}_{s\in S}$ commuting with the operations and relations in Σ. Σ-algebras, together with the Σ-homomorphisms form the category, $\mathbf{Alg}_\Sigma$.

In this paper we will have to relate algebras with different signatures. This is done through the notions of *forgetful functor* and *generalized algebra morphism*. In particular, as it is well known, every signature morphism $f: \Sigma_1 \to \Sigma_2$ has an associated forgetful functor $U_f: \mathbf{Alg}_{\Sigma_2} \to \mathbf{Alg}_{\Sigma_1}$. If f is a signature inclusion we may write $A_2|_{\Sigma_1}$ instead of $U_f(A_2)$. Then, a generalized algebra morphism (or, just, an algebra morphism) $h: A_1 \to A_2$, where $A_1 \in Alg_{\Sigma_1}$ and $A_2 \in Alg_{\Sigma_2}$ consists of a signature morphism $h_{Sig}: \Sigma_1 \to \Sigma_2$ and of a Σ_1-homomorphism $h_{Alg}: A_1 \to U_{h_{Sig}}(A_2)$. A generalized morphism h where both h_{Sig} and h_{Alg} are inclusions will be called an algebra embedding, or just an embedding. The class of all algebras, together with the generalized morphisms form the category, **Alg**. In [7] it is proved that **Alg** has pushouts. We will not consider any specific logic for writing the axioms associated to a model specification. It may be first-order logic, or some fragment of first-order logic, or a different kind of logic. We will only assume that if Form_Σ is the set of all Σ-formulas then:

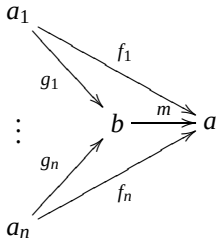
- There is a well-defined satisfaction relation between Σ-algebras and Σ-formulas, $\models \subseteq \mathbf{Alg}_\Sigma \times Form_\Sigma$.
- Signature morphisms induce formula translations, i.e. for every morphism $f: \Sigma_1 \to \Sigma_2$ there is an associated mapping $f^*: Form_{\Sigma_1} \to Form_{\Sigma_2}$.
- The so-called satisfaction property of institutions holds. This means that for each signature morphism $f: \Sigma_1 \to \Sigma_2$, each formula $\alpha \in Form_{\Sigma_1}$ and each algebra $A \in \mathbf{Alg}_{\Sigma_2}$ we have that $A \models f^*(\alpha)$ if and only if $U_f(A) \models \alpha$.

In these conditions, we may extend the previous constructions to deal with specifications (or software engineering models). In particular, given $SP = (\Sigma, Ax)$, we will denote

by $\mathbf{Alg}_{SP}$ the full subcategory of $\mathbf{Alg}_{\Sigma}$ consisting of all algebras satisfying the formulas in Ax. Also, we will consider that a specification morphism $f\colon (\Sigma_1, Ax_1) \to (\Sigma_2, Ax_2)$ is just a signature morphism $f\colon \Sigma_1 \to \Sigma_2$ such that for every $\alpha_1 \in Ax_1$, we have that $f^*(\alpha_1) \in Ax_2$. Specifications and specification morphisms form the category of specifications **Spec**. Moreover, every specification morphism $f\colon (\Sigma_1, Ax_1) \to (\Sigma_2, Ax_2)$ has an associated forgetful functor $U_f : \mathbf{Alg}_{(\Sigma_2, Ax_2)} \to \mathbf{Alg}_{(\Sigma_1, Ax_1)}$.

As said above, any software engineering model $SP = (\Sigma, Ax)$ is assumed to denote a class of instances, which are Σ-algebras that satisfy the axioms in Ax. However, not necessarily every algebra in $\mathbf{Alg}_{SP}$ needs to be considered a valid instance of SP. This means that we will consider that every software engineering model $SP = (\Sigma, Ax)$ denotes a category of algebras $\mathbf{Inst}_{SP}$, which is a subcategory of $\mathbf{Alg}_{SP}$. For example, from now on we will consider that every instance of a software engineering model extends a basic data algebra D. This means that given a model $SP = (\Sigma, Ax)$ we assume that $\Sigma_{data} \subseteq \Sigma$, where Σ_{data} is a given data signature. We also assume given a data algebra $D \in \mathbf{Alg}_{\Sigma_{data}}$ such that for each $A \in \mathbf{Inst}_{SP}$ we have $A|_{\Sigma_{data}} = D$. Moreover, if $h\colon A \to B$ in a homomorphism in $\mathbf{Inst}_{SP}$ then h is data preserving, which means that $h|_{\Sigma_{data}} : A|_{\Sigma_{data}} \to B|_{\Sigma_{data}}$ is the identity. In the same sense, we say that a signature morphism $f\colon \Sigma_1 \to \Sigma_2$ is data preserving if both signatures Σ_1 and Σ_2 include Σ_{data} and f is the equality when restricted to Σ_{data}. Similarly, a generalized algebra morphism $h\colon A_1 \to A_2$ is data preserving if h_{Sig} is data preserving and $h|_{\Sigma_{data}} : A_1|_{\Sigma_{data}} \to U_{h_{Sig}}(A_2)|_{\Sigma_{data}}$ is the identity.

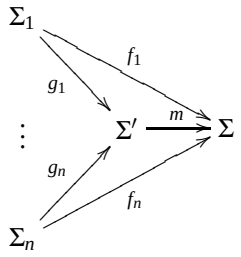
A property that we use to prove our results is pair factorization (actually, its general version, n-factorization). More precisely, n morphisms, $f_i\colon a_i \to c$, for $1 \leq i \leq n$, are *jointly epimorphic* if for all morphisms $f, g\colon c \to d$, we have that $f \circ h_i = g \circ h_i$, for every i, implies $f = g$. A category has the *n-factorization property* if for every family of morphisms $\{a_1 \xrightarrow{f_1} a, \ldots, a_n \xrightarrow{f_n} a\}$, with the same codomain a there exists an object b, a monomorphism m and a jointly surjective family of morphisms $\{a_1 \xrightarrow{g_1} b, \ldots, a_n \xrightarrow{g_n} b\}$ such that the diagram below commutes:



moreover if $\{f_1, \ldots, f_n\}$ are monomorphisms then so are $\{g_1, \ldots, g_n\}$. It is easy to prove the following proposition:

Proposition 1. ***(n-factorization)*** *The categories* **Sig**, **Spec**, $\mathbf{Alg}_{\Sigma}$, $\mathbf{Alg}_{SP}$, *and* **Alg** *have the n-factorization property, moreover if the morphisms f_i are data preserving so are the g_i and m.*

Proof. Given the signature morphisms $f_i\colon \Sigma_i \to \Sigma$, we define Σ' as the subsignature of Σ that includes all the sorts, operators and relation symbols in the images of the f_i. Then, we have that the diagram below commutes:



where m is the inclusion morphism. The proofs for **Spec**, $\mathbf{Alg}_{\Sigma}$, $\mathbf{Alg}_{SP}$, and **Alg** are similar. ■

In addition, we will assume that for any software engineering model *SP*, $\mathbf{Inst}_{SP}$ also has the *n*-factorization property.

3 Algebraic patterns and triple algebras

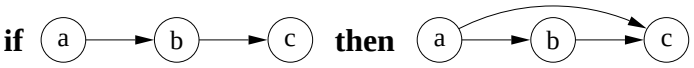
In this section we introduce the two basic constructions that are used to specify model transformations. First, we introduce the notion of algebraic patterns, as a generalization of the notion of graph constraints or graph conditions [9, 13, 15, 14, 22]. Then we present the notion of triple algebras, which is the corresponding generalization of the notion of triple graphs, introduced by Schürr [28].

3.1 Algebraic patterns

Graph constraints were introduced in the area of graph transformation, together with the notion of (negative) application conditions, as a form to limit the applicability of transformation rules [9, 13, 15, 14]. The underlying idea of a graph constraint is that it should specify that certain patterns must be present (or must not be present) in a given graph. However, later, several authors [16, 23, 25] have shown how graph constraints may be used for visual modelling. The simplest kind of constraint, which we called basic constraints in [23], is just a graph *C*. If *C* is considered to be a positive constraint then a graph *G* would satisfy *C* if it includes a copy of *C*. Conversely, if *C* is considered to be a negative constraint (or the negation of a positive constraint) then a graph *G* would satisfy *C* if it does not include any copy of *C*. For example, given the constraint *C*:

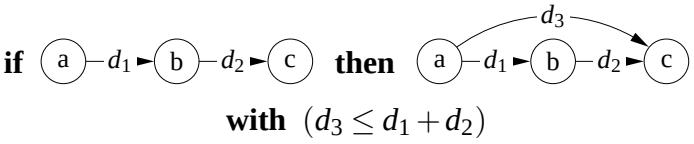
if the constraint is considered to be positive then it would specify that a given graph *G* should include at least two nodes with two edges connecting them. Conversely, if *C* is considered to be a negative constraint then it would specify that a given graph *G* should not include two different edges between any two nodes. A slightly more complex kind of graph constraints are graph monomorphisms (or, just, inclusions) $c : X \to C$, which we represent pictorially using an **if - then** notation, where the two graphs involved have

been labelled to implicitly represent the given monomorphism. These constraints are sometimes called conditional constraints and are a generalization of basic constraints: the constraint C is equivalent to the constraint $c : X \to C$ when X is the empty graph. If the constraint c is considered to be positive then it specifies that whenever a graph G includes (a copy of) the graph X it should also include (a copy of) its extension C. If c is considered to be a negative constraint then it would specify that the graph G includes a copy of the graph X but not a copy of its extension C. For instance, if the constraint c:



is a positive constraint then it would specify that a graph must be transitive, i.e. the constraint says that for every three nodes, a, b, c if there is an edge from a to b and an edge from b to c then there should be an edge from a to c. If c is negative then it would specify that a graph must not be transitive. In what follows, satisfaction of negative constraints is just the negation of satisfaction of positive constraints, and to avoid confusion between positive and negative constraint, we will write negative constraints using the negation symbol. This means that a constraint C will be assumed to be positive, while a constraint $\neg C$ will be assumed to be a negative constraint.

If we want to define constraints on attributed graphs, i.e. graphs that can *store* values from a given data algebra, in principle, we can use the same kind of constructions. This means that a graph constraint would be either a graph C or an inclusion $c : X \to C$, where X and C are attributed graphs. However, as discussed in [22], this poses some problems. On one hand, it may be difficult (if not impossible) to express constraints that include (or depend on) conditions on the data values. On the other, the formal definition of attributed graphs [8] is a bit complex. This causes that further technical development becomes complicated. To avoid this problems, in [22], a solution inspired in the area of Constraint Logic Programming is proposed. The idea is to separate the *graph part* in the constraints from the *data part*. This is done by considering that a graph constraint consists of a graph labelled with some variables (respectively, a morphism between labelled graphs) and a formula expressing a condition on these variables. For instance, the constraint below:



may express that, in a graph representing the distances between some cities, these distances must satisfy the so-called triangular property.

It must be mentioned that this separation of concerns, between the graph and the data part of a graph constraint, may also facilitate the construction of deductive and verification tools. In particular it would allow us to reuse existing powerful and efficient constraint solvers to deal with the data part of constraints.

Algebraic patterns, as presented in this paper, are inspired and generalize attributed graph constraints. The basic idea is similar: an algebraic pattern consists of an algebra with variables (instead of data) (or an embedding between algebras with variables) and a formula expressing some conditions on these variables. This is defined as follows: given a (data sorted) set of variables X, first, we define a signature of variables Σ_X whose only operation symbols are the variables (seen as constants); next, for each signature Σ extending Σ_{data} we define the signature $\Sigma(X)$, obtained replacing in Σ the operators and relation symbols in Σ_{data} by the variables in X; then, we define an algebra of variables $\aleph_X$ whose only elements are the variables in X; finally, a pattern will just be an algebra extending $\aleph_X$ or an embedding over algebras extending $\aleph_X$, together with some Σ_{data}-formula on the variables. From now on we will assume that the given sets of variables are finite.

Definition 1 (Signature of variables, Algebra of variables). *Given a data signature Σ_{data} and an S_{data} sorted finite set of variables X, we define the signature of variables over X, Σ_X, as the signature $(S_{data}, X, \emptyset)$. Given a signature Σ extending Σ_{data}, the extension of Σ over X, $\Sigma(X)$ is defined by the pushout in the diagram below:*

$$\begin{array}{ccc} (S_{data}, \emptyset, \emptyset) & \longrightarrow & (S_{data}, X, \emptyset) \\ \downarrow & po & \downarrow \\ \Sigma \setminus \Sigma_{data} & \longrightarrow & \Sigma(X) \end{array}$$

where $\Sigma \setminus \Sigma_{data}$ denotes the signature where all the operators and relation symbols (but no the sorts) in Σ_{data} have been removed from Σ.

We also define the algebra of variables over X, $\aleph_X$, as the term algebra over Σ_X, T_{Σ_X}.

Notice that every generalized morphism from the algebra of variables $\aleph_X$ into the data algebra D can be seen as a variable assignment, and vice versa.

Definition 2 ($\Sigma(X)$-algebras with variables, Algebraic patterns). *Given an S_{data} sorted set of variables X, and a signature Σ extending Σ_{data}, a $\Sigma(X)$-algebra with variables in X is a finite $\Sigma(X)$-algebra extending $\aleph_X$. Given a $\Sigma_1(X)$-algebra A_1 and a $\Sigma_2(X)$-algebra A_2, a generalized morphism with variables $h : A_1 \to A_2$ is a generalized morphism such that $h_{Alg|\Sigma_X}$ is the equality. As before, generalized morphisms with variables whose signature and algebra parts are inclusions are called embeddings.*

A basic algebraic pattern P, is a 3-tuple (X, C, α), where X is a data sorted set of variables, C is a $\Sigma(X)$-algebra with variables and α is a Σ_{data}-formula with free variables in X.

A conditional algebraic pattern CP is a 3-tuple (X, c, α), where X is a data sorted set of variables, $c : C_1 \to C_2$ is an embedding with variables, C_1 and C_2 are $\Sigma_1(X)$ and $\Sigma_2(X)$-algebras, respectively, and α is a Σ_{data}-formula with free variables in X.

For instance, let us consider that a graph (an attributed graph) is an algebra over a signature with two sorts, $Nodes$ and $Edges$ (in addition to the data sorts), and two operations $source: Edges \to Nodes$ and $target: Edges \to Nodes$ (in addition to the

operations in Σ_{data}). In that case, the above examples of graph constraints could be considered examples of algebraic patterns.

Satisfaction of algebraic patterns is just the straightforward generalization of graph constraint satisfaction. In particular, Given a $\Sigma(X)$-algebra A (positively) satisfies the basic pattern (C,α) if there is a generalized monomorphism $h : C \to A$ such that the data algebra D satisfies the formula α when the variables in X have been replaced by their images in h. Satisfaction of conditional patterns is the corresponding generalization. Satisfaction Negative patterns is just the negation of positive satisfaction.

Definition 3 (Satisfaction of algebraic patterns). *A Σ-algebra A satisfies the basic pattern (X,C,α), denoted $A \models (X,C,\alpha)$ if there is a generalized monomorphism $h : C \to A$ such that $D \models_{h_{Alg}|_{\Sigma_X}} \alpha$.*

A Σ-algebra A satisfies the conditional algebraic pattern $(X,c : C_1 \to C_2,\alpha)$ if for each generalized monomorphism $m : C_1 \to A$ such that $D \models_{m_{Alg}|_{\Sigma_X}} \alpha$ there is a generalized monomorphism $m' : C_2 \to A$ such that $m = m' \circ c$.

3.2 Triple algebras and triple patterns

Triple graphs were defined by Schürr [28] to describe model transformations when models and instances are represented as graphs. Instead of considering that a model transformation is just characterized by the source and target model, Schürr considered that it is very important to establish a connection between the corresponding elements of the source and target graph. The way to do this is using an intermediate graph, the *connection graph* and one mapping from this connection graph into each of the source and target graphs. Here, we use the same idea, but using algebras instead of graphs.

Definition 4 (Triple algebras and Morphisms). *A triple algebra $TrA = (TrA_S \xleftarrow{c_S} TrA_C \xrightarrow{c_T} TrA_T)$ (or just $TrA = \langle TrA_S, TrA_C, TrA_T\rangle$ if c_S and c_T may be considered implicit) consists of three algebras TrA_S, TrA_C, and TrA_T, called the source, connection and target algebras, respectively, and two generalized morphisms c_S and c_T. We say that TrA extends the data algebra D if the three algebras TrA_S, TrA_C, TrA_T extend D and theΣ_{data} reduct of the morphisms c_S and c_T coincides with the identity. A triple algebra morphism $m = (m_S, m_C, m_T) : TrA^1 \to TrA^2$ consists of three generalized morphisms m_S, m_C , and m_T such that $m_S \circ c_S^1 = c_S^2 \circ m_C$ and $m_T \circ c_T^1 = c_T^2 \circ m_C$. In addition, m is data preserving if theΣ_{data} reduct of the morphisms m_S, m_C and m_T coincides with the identity.*

Given a triple algebra TrA, we write $TrA|_K$ for $K \in \{S,C,T\}$ to refer to a triple algebra where only the K algebra is present and the other two components are the empty algebra over the empty signature, i.e., $TrA|_S = \langle TrA_S, \emptyset, \emptyset\rangle$, and given a triple algebra morphism $h : TrA_1 \to TrA_2$ we also write $h|_K : TrA_1|_K \to TrA_2|_K$ to denote the morphism whose K-component coincides with h_K and whose other two components are the empty morphism between empty algebras. Finally, given TrA, we write i_G^K to denote the inclusion $i_{TrA}^K : TrA|_K \to TrA$, where the K-component is the identity.

Triple algebras form the category **TrAlg**, which can be formed as the functor category $\mathbf{Alg}^{\cdot\leftarrow\cdot\rightarrow\cdot}$.

In the same way that algebraic patterns describe properties that may be satisfied by an algebra, triple algebraic patterns describe properties that may be satisfied by a triple algebra. More precisely, first we may define the notions of triple algebra with variables and triple morphism with variables:

Definition 5 (Triple algebras and Morphisms with variables). *A triple algebra with variables in X, $TrA = (TrA_S \xleftarrow{c_S} TrA_C \xrightarrow{c_T} TrA_T)$ is a triple algebra where TrA_S, TrA_C and TrA_T are algebras with variables in X and c_S and c_T are generalized morphisms with variables.*

A triple algebra morphism $m = (m_S, m_C, m_T)\colon TrA^1 \to TrA^2$ is a triple morphism such that m_s, m_c and m_T are morphisms with variables.

It must be noted that if $m = (m_S, m_C, m_T)\colon TrA^1 \to TrA^2$ is a data preserving triple morphism, where $TrA^1 = (TrA^1_S \xleftarrow{c^1_S} TrA^1_C \xrightarrow{c^1_T} TrA^1_T)$ is a triple algebra with variables in X and $TrA^2 = (TrA^2_S \xleftarrow{c^2_S} TrA^2_C \xrightarrow{c^2_T} TrA^2_T)$ is a triple algebra that extends D, then m maps every variable in X to a unique data value. In particular, if x is a variable in X then x is a constant in the signatures of TrA^1_S, TrA^1_C, and TrA^1_T. Now, let $d_1 = m_S(x_{TrA^1_S})$, $d_2 = m_C(x_{TrA^1_C})$, and $d_3 = m_T(x_{TrA^1_T})$. Let us prove that $d_1 = d_2$:

$d_1 = m_S(x_{TrA^1_S}) = m_S(c^1_S(x_{TrA^1_C})) = c^2_S(m_C(x_{TrA^1_C})) = c^2_S(d_2)$. But, since c^2_S is assumed to preserve data, this means that $d_1 = d_2$. The proof that $d_2 = d_3$ is similar. Therefore, we may identify m with a variable assignment.

Now, we extend our notion of algebraic pattern to the notion of triple algebraic pattern, together with the corresponding definition of satisfaction:

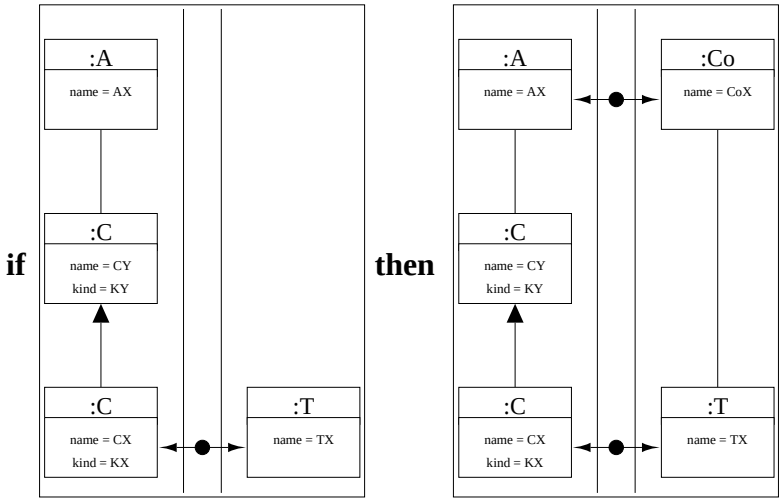
Definition 6 (Triple algebraic patterns). *A basic triple algebraic pattern P, is a 3-tuple (X, TrC, α), where X is a data sorted set of variables, TrC is a triple algebra with variables in X and α is a Σ_{data}-formula with free variables in X.*

A conditional algebraic pattern CP is a 3-tuple (X, c, α), where X is a data sorted set of variables, c is a triple morphism with variables and α is a Σ_{data}-formula with free variables in X.

Definition 7 (Satisfaction of triple algebraic patterns). *A triple algebra TrA that extends D satisfies the basic triple pattern (X, TrC, α), denoted $A \models (X, TrC, \alpha)$ if there is a data preserving triple monomorphism $h : TrC \to TrA$ such that $D \models_h \alpha$.*

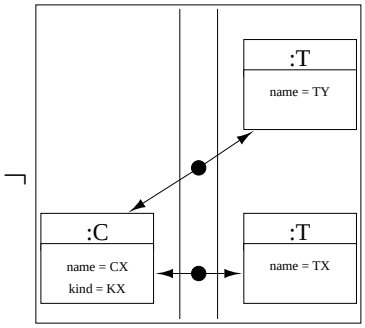
A triple algebra TrA that extends D satisfies the conditional triple algebraic pattern $(X, c : TrC_1 \to TrC_2, \alpha)$, if for each data preserving triple monomorphism $m : TrC_1 \to TrA$ such that $D \models_h \alpha$ there is a data preserving triple monomorphism $m' : TrC_2 \to TrA$ such that $m = m' \circ c$.

Example 1. A standard example in model transformation is the transformation of class diagrams into relational schemas [26]. According to this transformation (persistent) classes are transformed into tables and attributes are transformed into columns, Then, if a class c_1 is a subclass of a class c_2 and c_2 has a certain attribute at, we know that c_1 inherits that attribute. This means that the table associated to c_1 should include a column associated to at. The pattern specifying this property, which this transformation must satisfy, is depicted below.



More precisely, this pattern specifies that whenever a triple algebra includes a class CX, which is a subclass of CY, and such that CY has an associated attribute X and CX is transformed into the table TX then that triple algebra must also include a column CoX, which is associated to TX and is the transformation of AX. It may be noted that the above pattern does not include any formula on the variables CX, KX, KY, CY, AX, TX and CoX. The reason is that in this case there is no specific condition needed. Hence the formula is empty.

Another property that our transformation must satisfy is that a given class should not be transformed into two different tables. This may be specified by the negative pattern below:



In [23, 22] it is shown how we can reason with graph constraints. In particular, sound and complete proof systems are presented which apply to constraints over a large class of graphical structures, including triple graphs. We believe that the results presented in these papers could also be applied to the case of algebraic patterns and triple algebraic patterns in the case where the algebras involved in the patterns are finite.

4 Specification of model transformations by triple patterns

A model transformation is a procedure that given instances of a source model yields instances of a target model. This procedure may be described algorithmically or declaratively. In this work we are concerned with the declarative description of graph transformation. In our context, specifying a model transformation means describing a class of triple algebras $\langle TrA_S, TrA_C, TrA_T \rangle$, where TrA_S is an instance of the source model

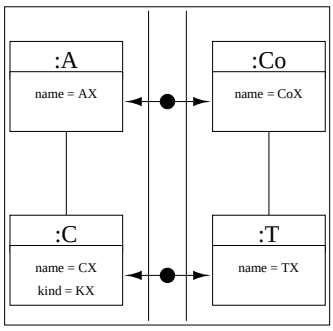
and TrA_T is an instance of the target model. In particular, we could use triple algebraic patterns, as described in the previous section, to write this kind of specifications. More precisely, this means considering that a model transformation specification would be a set of positive and negative patterns (or, more generally, a set of axioms consisting of logical formulas whose atomic components are triple algebraic patterns). Then, a specification would denote, as usual, the class of all triple algebras satisfying these axioms. In that context, techniques similar to those developed in [23, 22] would allow us, not only to reason about our specifications, including checking its consistency, but also to build minimal triple algebras satisfying the specification. Which means that we could have a way of automatically implementing the model transformation. However, in our opinion, an approach like the one introduced in [6] and further developed in [24, 12], also based on the use of triple patterns, may be more appropriate for this purpose. On one hand, that approach is in a way close to the declarative fragment of OMG's standard language QVT (QVT-relational) [26] and may be found easier and more intuitive to use for defining specific transformations. The reason is that, even if the approach is essentially declarative, in a sense reflect how one would construct the transformation. On the other hand, and partly as a consequence of the previous reason, that approach seems more appropriate for its (automatic) implementation. In particular, in [24], we have shown that, in the case of triple graphs and triple graph patterns, a model transformation specification in the sense of [6] may be converted into a finitely terminating graph transformation system where all the models (in the logical sense) of the specification are terminal graphs of the transformation system.

Patterns, as presented in [6, 24, 12], from now on called *transformation patterns*, have a similar syntax but different semantics to the patterns that we present in the above section. The basic idea of transformation patterns is (in a way) to see them as tiles that have to "cover" a given source model, perhaps with some overlapping. The target model obtained by gluing the target parts of these patterns is the result of the transformation. In particular, this is formally stated saying that a triple instance (forward)[3] satisfies a transformation pattern if whenever the source part of the instance embeds the source part of the pattern then the whole instance should embed the pattern. Let us provide some intuition with an example:

Example 2. Let us suppose that we want to specify the transformation of class diagrams into relational schemas [26]. According to this transformation, persistent classes are transformed into tables and their associated attributes transformed into columns of these tables. This may be specified by the pattern below

[3] Triple patterns may be considered to specify bidirectional transformations, i.e. source-to-target transformations and target-to-source transformations. In this sense, there is a corresponding notion of backward satisfaction. Actually, most of the associated notions and constructions that are presented below have a corresponding backward version.

(1)

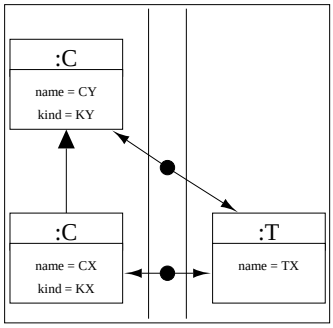


with $KX = persistent \wedge CoX = "\_" + AX$

Then a triple instance would satisfy this pattern if, whenever there is a class in the source instance whose kind is persistent, and this class has an associated attribute, then the triple instance also includes a table and an associated column which are the transformation, respectively, of the class and the attribute. In this case, the formula in the pattern states, on one hand that the kind of the class must be persistent and that the name of the column is obtained by adding an underline character in front of the name of the attribute.

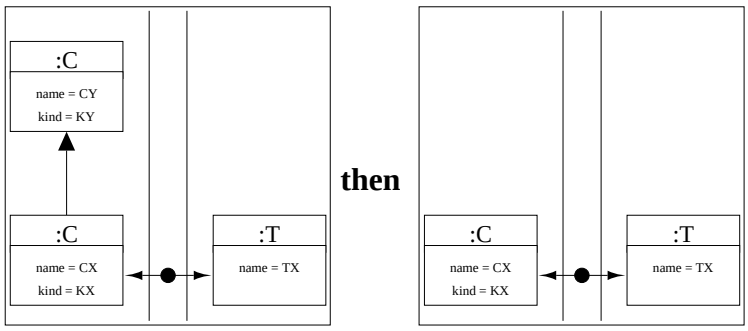
Another pattern may specify that subclasses and parent classes are transformed into the same table:

(2)



with $KX = persistent$

Transformation patterns may also have a condition. In this case, the condition relates to the context of the pattern and is always negative. It plays a similar role to negative application conditions in graph transformation. Let us see an example:

(3) if ¬ 

with $KX = persistent \wedge TX = CX$

The above pattern specifies that every persistent class if it does not have a parent class then it must be transformed into a table with the same name as the class. Notice that the condition includes the conclusion of the pattern. The reason is that the condition always refers to the context of the pattern.

In addition, negative basic patterns like the one presented in Example 1 may be used when specifying a transformation to describe relationships that should not occur between the source and target models.

Definition 8 (Transformation patterns, transformation specifications). *A transformation pattern* $TP = \langle X, \{TrC \xrightarrow{n_j} N_j\}_{j \in J}), TrC, \alpha \rangle$ *consists of*

- *A data sorted set of variables* X.
- *The postcondition, given by the triple algebra TrC with variables in* X.
- *The negative preconditions, given by the embeddings with variables* $TrC \xrightarrow{n_j} N_j$, *for each* $j \in J$.
- *A data condition given by* α, *which is a* Σ_{data}*-formula with free variables in* X.

A transformation specification TSP is a finite set of transformation patterns and negative algebraic basic patterns.

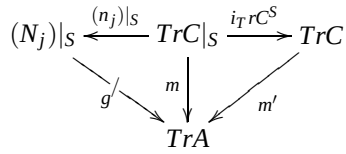
Then, satisfaction is defined as described in the example. More precisely, a triple algebra TrA satisfies a transformation pattern $TP = \langle X, \{TrC \xrightarrow{n_j} N_j\}_{j \in J}), TrC, \alpha \rangle$ if whenever there is a monomorphism m from TrC_S into the source of TrA, the preconditions are satisfied by m and the formula α is satisfied by the data algebra with respect to the assignment defined by m, then there is a monomorphism from TrC into TrA extending m. And a monomorphism m of TrC_S in TrA_S satisfies a precondition $TrC \xrightarrow{n_j} N_j$ if there is no embedding m' of $(N_j)_S$ in TrA_S such that m' extends m. Then, we may consider that the semantics of a transformation specification TSP is the class of all triple algebras that satisfy the patterns in TSP.

Definition 9 (Satisfaction of transformation patterns).

A monomorphism $m : TrC|_S \to TrA$ *satisfies a negative precondition* $TrC \xrightarrow{n_j} N_j$, *denoted* $m \models TrC \xrightarrow{n_j} N_j$ *if there does not exist a monomorphism* $g : (N_j)|_S \to TrA$ *such that* $m = g \circ (n_j)|_S$.

A triple algebra TrA satisfies a transformation pattern $TP = \langle X, \{TrC \xrightarrow{n_j} N_j\}_{j \in J}), TrC, \alpha \rangle$ *, denoted* $TrA \models TP$, *if for every monomorphism* $m : TrC|_S \to TrA$, *such that for every* j

in J $m \models TrC \xrightarrow{n_j} N_j$ *and such that* $D \models_m \alpha$*, there exist a monomorphism* $m' : TrC \to TrA$ *such that* $m = m' \circ i_{TrC}^S$*:*



A triple graph TrA satisfies a negative basic pattern $\neg(X, TrC, \alpha)$*, denoted* $TrA \models \neg(X, TrC, \alpha)$ *if there is no injective morphism* $h : TrC \to TrA$ *such that* $D \models_h \alpha$*.*

Then we could define the semantics of a transformation specification TSP as the class of all triple algebras that satisfy the patterns in TSP. However, in our opinion, as argued in [24], this semantics would be too loose. In particular, it would include triple algebras whose target and connection part cannot be *generated* by the patterns. For instance, models whose target part includes some kinds of elements of a given type not mentioned in the patterns. We think that restricting our attention to this kind of *generated models* is reasonable in this context. This is similar to the "No Junk" condition in algebraic specification.

Definition 10 (TSP-generated triple algebras). *Given a pattern specification* TSP*, a triple algebra* TrA *is* TSP*-generated if there is a finite family of transformation patterns* $\{\mathcal{TP}_k\}_{k\in K}$*, with* $TP_k = \langle X_k, \{TrC_k \xrightarrow{n_jk} N_jk\}_{jk\in J_k}), TrC_k, \alpha_k\rangle$*, and a family of monomorphisms* $\{TrC_k \xrightarrow{f_k} TrA\}_{k\in K}$ *such that every* f_k *forward satisfies all the preconditions in* $\mathcal{N}_k$*,* $D \models_{f_k} \alpha_k$*, and* $f_1, \ldots, f_n$ *are jointly surjective.*

Then, we can finally define the semantics of a transformation specification TSP:

$$Sem(TSP) = \{TrA \mid \forall P \in TSP\ TrA \models P \text{ and TrA is a TSP-generated triple algebra}\}$$

5 Verification of model transformations

In [2] a formal framework for the verification of model transformations is proposed considering that the source and target models are, in some sense, semantically equivalent. This may be reasonable for some kinds of model transformations, like refactoring [27], or the example considered in [2] (again, the class diagrams-relational schemas transformation). However, this is not so reasonable in other classes of transformations: consider for instance (as an extreme case) a transformation from sequence diagrams into class diagrams. In our opinion, a transformation may be considered correct if some observable properties of the source model remain (in some sense) invariant in the target model. For instance, in the sequence diagrams-class diagrams transformation we may only want to preserve typing. In the class diagrams-relational schemas transformation we may want to preserve the class-attribute association (including the association with inherited attributes). And in the case of some transformation between behaviour models

we may want to preserve some behaviour properties like deadlock-freeness. This is the case when the main reason for applying the transformation is to use some verification tools which are available only for the class of target models.

As a consequence, we believe that the verification of a model transformation is only possible if the properties that characterize its correctness are given a priori. However, it may be not straightforward to state that a certain property remains invariant after a given transformation. The reason is that the source and target models may be quite different, so the corresponding properties may also look quite different. In this sense, we believe that the use of triple algebras and triple patterns to model transformations may help. In particular, we think that the possibility of dealing together with the source and target models and he connections between them helps in expressing simultaneously that a given property holds in the source model if and only in the corresponding property holds in the target model. For instance, the positive algebraic pattern presented in Example 1 can be seen as expressing that the class-attribute association remains invariant in the class diagrams-relational schemas transformation. This means that to verify a model transformation we must first provide a *verification specification*. More precisely, verification specifications should consist only of positive patterns. There are two reasons for this. On one hand, typically properties expressing invariants are, in this context, positive properties. On the other, negative patterns play a better role in transformation specifications, describing what kind of situations are forbidden. Actually, if the model transformation is going to be implemented by graph transformation (or a similar technique) as in [24] then the negative patterns may be embedded into the transformation rules as negative application conditions. Then a model transformation specified by TSP would be correct with respect to a verification specification VSP if every triple algebra in the semantics of TSP satisfies VSP:

Definition 11 (Verification specification, Correctness of transformation specifications). *A verification specification VSP is a finite set of triple algebraic patterns. Then a transformation specification TSP is correct with respect to VSP if for every triple algebra $TrA \in Sem(TSP)$ and every triple $TrP \in VSP$ $TrA \models TrP$.*

Now, the next question is, given specifications TSP and VSP, how can we verify that TSP is correct with respect to VSP. If we have a method to generate all the triple algebras in the semantics of TSP, as in [24], an obvious, (but not very efficient) way would be to generate all these triple algebras and check that they satisfy the verification conditions. This approach may seem not very elegant but one could devise a deductive method based on these ideas. However, the main problem of this kind of solution is that a procedure based on this approach would only be a semi-decision procedure that would provide an answer in finite time only if TSP is incorrect, since normally there would be an infinite number of triple algebras in $Sem(TSP)$.

Instead, in what follows, we describe the basis of a finitely terminating procedure that, if the answer is positive, ensures that a given transformation specification is correct with respect to VSP. The basic idea is quite simple. Since the triple algebras in $Sem(TSP)$ are TSP-generated, this means that each triple algebra in $Sem(TSP)$ can be seen as the gluing of some of the transformation patterns in TSP. This means that we can be sure that TSP is correct with respect to VSP, if we are able to prove that each

possible gluing of the transformation patterns in TSP, which satisfies the negative patterns in TSP, also satisfies the patterns in VSP. In particular, the key aspect is that, even if the number of possible ways of gluing the patterns in TSP is also infinite, it is enough to check a finite number of them to ensure correctness. In particular, it is enough to check the gluings which are *minimal*. But before describing this technique we first need some auxiliary definitions. The first one describes a notion of satisfaction between triple patterns (more precisely, satisfaction of arbitrary triple patterns by basic patterns). The definition is quite close to the definition of satisfaction of triple patterns by triple algebras. The main difference is on how we deal with the formulas involved in the patterns.

Definition 12 (Satisfaction of triple patterns by basic patterns). *A basic triple algebraic pattern $P = (X, TrC, \alpha)$ satisfies a conditional pattern $CP = (X', c : TrC_1 \rightarrow TrC_2, \alpha')$, where X and X' are disjoint, denoted $P \models CP$ if for each triple monomorphism $m : TrC_1 \rightarrow TrC$ such that $\alpha \wedge \alpha' \wedge eq(m)$ is satisfiable in D there is a triple monomorphism $m' : TrC_2 \rightarrow TrC$ such that $m = m' \circ c$, where eq(m) a conjunction of equations $x = m(x)$ for each x in X and where $m(x)$ denotes the image of x in X' through m.*

The second definition that we need is the notion of gluing of patterns.

Definition 13 (Gluing of patterns). *A basic triple algebraic pattern $P = (X, TrC, \alpha)$ is the gluing of the finite family of transformation patterns $\{\mathcal{TP}_k\}_{k \in K}$, with $TP_k = \langle X_k, \{TrC_k \xrightarrow{n_j k} N_j k\}_{jk \in J_k}), TrC_k, \alpha_k \rangle$, where all the X_k are pairwise disjoint, via a family of monomorphisms $\{TrC_k \xrightarrow{f_k} TrC\}_{k \in K}$ if every f_k satisfies all the preconditions in $\mathcal{N}_k$, $f_1, \ldots, f_n$ are jointly surjective, and $\alpha = \bigwedge_k \alpha_k \wedge eq(\{f_k\}_{k \in K})$, where $eq(\{f_k\}_{k \in K})$ denotes the conjunction of equations $x_i = x_j$ with $x_i \in X_i$, $x_j \in X_j$ and $f_i(x_i) = f_j(x_j)$.*

The last definition that we need is the definition of minimal gluing. The idea is that a TSP-generated pattern $P = (X, TrC, \alpha_1)$ is minimal with respect to a monomorphism $h : TrC' \rightarrow TrC$ if all the patterns that participate in the gluing have some element in the image of h and if, in addition, if the elements in the image of h covered by a pattern are not covered already by the other patterns.

Definition 14 (Minimal gluing of patterns). *A basic triple algebraic pattern $P = (X, TrC, \alpha)$ which is the gluing of the family $\{\mathcal{TP}_k\}_{k \in K}$ via a family of monomorphisms $\{TrC_k \xrightarrow{f_k} TrA\}_{k \in K}$ is minimal with respect to the monomorphism $h : TrC_0 \rightarrow TrC$ if there is no pattern $P' = (X', TrC', \alpha')$ which is the gluing of the family $\{\mathcal{TP}_j\}_{j \in J}$ via $\{TrC_j \xrightarrow{f'_j} TrC\}_{j \in J}$, with $J \subset K$, such that there are two monomorphisms $m : TrC_0 \rightarrow TrC'$ and $m' : TrC' \rightarrow TrC$ such that for every $i \in J \cap K$, $f'_i = m' \circ f_i$, and moreover $m' \circ m = h$.*

Now, we are ready to present the two results that are the basis for a procedure for checking the correctness of a transformation specification.The first result ensures the termination of the procedure:

Theorem 1. *Given a finite transformation specification TSP, and given a finite triple algebra with variables TrC_0, there is a finite number of monomorphisms $h : TrC_0 \to TrC$ such that there is a pattern $P = (X, TrC, \alpha)$ which is the gluing of a family of patterns in TSP and which is minimal with respect to h.*

Proof. Since TrC_0 is finite, it has a finite number of elements, let us say n. This means that if P is minimal with respect to a monomorphism h, then P is the gluing of at most n patterns. In addition, we know that if $TP = \langle X, \{TrC' \xrightarrow{n_j} N_j\}_{j\in J}), TrC', \alpha\rangle$ is a pattern in TSP then TrC' is a finite algebra. This means that there is a finite number of patterns P which are the gluing of n or less patterns in TSP. Finally, since TrC_0 and TrC are both finite, there is a finite number of monomorphisms from TrC_0 into TrC. ■

The second result states that if all the minimal patterns obtained by overlapping transformation patterns in TSP satisfy a verification pattern then all triple algebras in $Sem(TSP)$ also satisfy the verification pattern.

Theorem 2. *Given a finite transformation specification TSP, if for every triple pattern $CP = (X', c : TrC_1 \to TrC_2, \alpha')$ in the verification specification VSP, every monomorphism $h : TrC_0 \to TrC$ such that there is a pattern $P = (X, TrC, \alpha)$ which is the gluing of a family of patterns in TSP and which is minimal with respect to h and such that TrC satisfies the negative constraints in TSP we have that $P \models CP$ then TSP is correct with respect to VSP.*

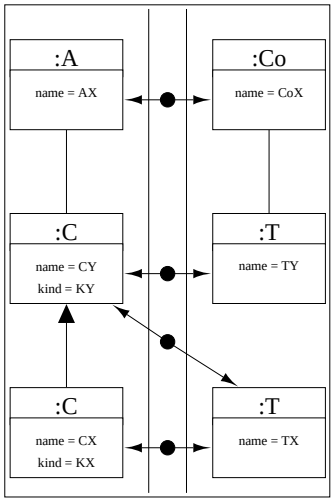
Proof. (Sketch)

Let us suppose that TSP is not correct with respect to VSP. This means that there is a triple algebra TrA in $Sem(TSP)$ that does not satisfy some triple pattern $CP = (X', c : TrC_1 \to TrC_2, \alpha')$ in VSP. Since $TrA \in Sem(TSP)$ then TrA is $TSP-generated$. This implies that there is a finite family of transformation patterns $\{\mathcal{TP}_k\}_{k\in K}$, with $TP_k = \langle X_k, \{TrC_k \xrightarrow{n_j k} N_j k\}_{jk\in J_k}), TrC_k, \alpha_k\rangle$, and a family of monomorphisms $\{TrC_k \xrightarrow{f_k} TrA\}_{k\in K}$ such that every f_k forward satisfies all the preconditions in $\mathcal{N}_k$, $D \models_{f_k} \alpha_k$, and $f_1, \ldots, f_n$ are jointly surjective. But then we can build a pattern $P = (X, TrC, \alpha)$ by gluing the family $\{\mathcal{TP}_k\}_{k\in K}$ via some monomorphisms $\{TrC_k \xrightarrow{f_k} TrC\}_{k\in K}$ in such a way that there is a monomorphism $g : TrC \to TrA$ g_s for every non-data sort s. Moreover, if TrA satisfies all the negative constraints in SP then P also satisfies all the negative constraints in TSP and if TrA does not satisfy CP then P neither satisfies CP.

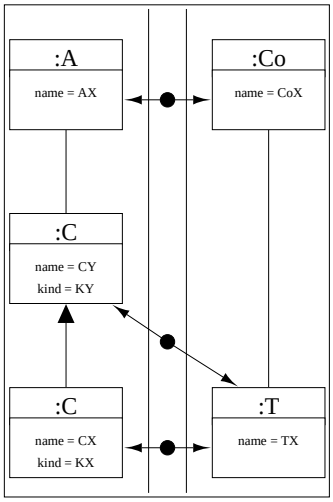
Now, if P does not satisfy CP this means that there is a monomorphism $m : TrC_1 \to TrC$ such that $\alpha \wedge \alpha' \wedge eq(m)$ is satisfiable in D but there is not a monomorphism $m' : TrC_2 \to TrC$ such that $m = m' \circ c$. Now, if P is minimal with respect to m we have proved the theorem. If P is not minimal with respect to h then let $P' = (X', TrC', \alpha')$ be obtained by the gluing of a family of $\{\mathcal{TP}_j\}_{j\in J}$, with $J \subset K$ via $\{TrC_j \xrightarrow{f'_j} TrC\}_{j\in J}$ and such that there are monomorphisms $h : TrC_0 \to TrC'$ and $h' : TrC' \to TrC$ such that for every $i \in J \cap K$, $f'_i = h \circ f_i$, and moreover $h' \circ h = m$. Then it is straightforward to see P' satisfies all the negative constraints in TSP and does not satisfy CP. ■

Example 3. Let us now prove that the transformation from class diagrams to relational schemas consisting of the transformation patterns presented in Example 2 together with the negative pattern presented in Example 1 satisfies the conditional pattern in Example 1 (let us call it *Attribute inheritance*). First of all, we can see that the condition in that pattern cannot be matched to the postcondition of any of the transformation patterns in Example 2.

In this case, the only minimal patterns that can match the condition of the *Attribute inheritance* pattern are obtained by gluing the transformation patterns (1) and (2) and are displayed below:



with $KX = persistent \wedge$
$\wedge\, KY = Persistent \wedge$
$\wedge\, CoX = "\_" + AX$



with $KX = persistent \wedge$
$\wedge\, KY = Persistent \wedge$
$\wedge\, CoX = "\_" + AX$

Now, the pattern on the left violates the negative pattern that specifies that one class cannot have two associated tables. Therefore, we can discard it. Then, pattern on the right is the only minimal gluing that satisfy the negative condition on the transformation specification. But this pattern satisfies satisfies the *Attribute inheritance* pattern. Therefore according to Theorem 2 the transformation specification is correct with respect to the *Attribute inheritance* pattern.

6 Conclusion and Future Work

In this paper we have discussed what does it mean to verify a model transformation, and we have proposed a specific method for doing so. More precisely, first, we have presented a method for specifying model transformations that can be considered a generalization of the method presented in [6, 12] and, then, we have presented a method to prove the correctness of a model transformation specification.

There are two main lines of work that should be pursued in the future. The first one has to do with refining and implementing the work presented in this paper. In particular, before implementing these ideas, it would be needed to present a specific algorithm for checking the correctness of a transformation based on the result presented in Theorem 2. Then, for the implementation there are two possibilities. On one hand, we could try to implement this approach using a tool like Maude [5] that would allow us to directly work on algebras. Or, alternatively, we could specialize our approach to the case of graphs, i.e. as in [6, 12], and use some graph transformation tool.

The second line of work has to do with the kind of models considered. As said above, in this work we consider only the transformation of structural models. This means that the ideas presented may have limited interest for the verification of transformation of behavioural models. In that case, one is not so much interested in knowing that each instance of the source model is correctly transformed into an instance of the target model, but that some observable properties of the overall behaviour of the source model remain invariant are also satisfied by the target model. This makes the problem technically very different.

References

1. A. Boronat, R. Heckel, and J. Meseguer. Rewriting logic semantics and verification of model transformations. In M. Chechik and M. Wirsing, editors, *Fundamental Approaches to Software Engineering, 12th International Conference, FASE 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, volume 5301 of *LNCS*, pages 18–33. Springer, 2009.
2. A. Boronat, A. Knapp, J. Meseguer, and M. Wirsing. What is a multi-modelling language? In *WADT 2008*, 2008.
3. A. Boronat and J. Meseguer. An algebraic semantics for mof. In J. L. Fiadeiro and P. Inverardi, editors, *Fundamental Approaches to Software Engineering, 11th International Conference, FASE 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008*, volume 4961 of *Lecture Notes in Computer Science*, pages 377–391. Springer, 2008.
4. A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and proof in membership equational logic. *Theor. Comput. Sci.*, 236(1-2):35–132, 2000.
5. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.
6. J. de Lara and E. Guerra. Pattern-based model-to-model transformation. In *Proc. ICGT'08*, volume 5214 of *LNCS*, pages 426–441. Springer, 2008.
7. H. Ehrig, M. Baldamus, and F. Orejas. Amalgamation and extension in the framework of specification logics and generalized morphisms. *Bulletin of the EATCS*, 44:129–143, 1991.
8. H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of algebraic graph transformation*. Springer-Verlag, 2006.
9. H. Ehrig and A. Habel. Graph grammars with application conditions. In G. Rozenberg and A. Salomaa, editors, *The Book of L*, pages 87–100. Springer, 1986.
10. J. A. Goguen and R. M. Burstall. Institutions: Abstract model theory for specification and programming. *J. ACM*, 39(1):95–146, 1992.

11. J. A. Goguen, J.-P. Jouannaud, and J. Meseguer. Operational semantics for order-sorted algebra. In W. Brauer, editor, *Automata, Languages and Programming, 12th Colloquium*, volume 194 of *Lecture Notes in Computer Science*, pages 221–231. Springer, 1985.
12. E. Guerra, J. de Lara, and F. Orejas. Pattern-based model-to-model transformation: Handling attribute conditions. In *ICMT 2009*, LNCS. Springer, accepted.
13. A. Habel, R. Heckel, and G. Taentzer. Graph grammars with negative application conditions. *Fundamenta Informatica*, pages 287–313, 1996.
14. A. Habel and K.-H. Pennemann. Correctness of high-level transformation systems relative to nested conditions. *Math. Struct. in Comp. Sc.*, To appear.
15. R. Heckel and A. Wagner. Ensuring consistency of conditional graph rewriting - a constructive approach. *ENTCS*, 2, 1995.
16. M. Koch, L. V. Mancini, and F. Parisi-Presicce. Graph-based specification of access control policies. *J. Comput. Syst. Sci.*, pages 1–33, 2005.
17. A. Königs and A. Schürr. Tool integration with triple graph grammars - a survey. *ENTCS*, 148(1):113–150, 2006.
18. T. Mens and P. V. Gorp. A taxonomy of model transformation. *Electr. Notes Theor. Comput. Sci.*, 152:125–142, 2006.
19. P. D. Mosses. Unified algebras. In *ADT*, 1988.
20. P. D. Mosses. Unified algebras and institutions. In *LICS 1989*, pages 304–312, 1989.
21. P. D. Mosses. Unified algebras and modules. In *Sixteenth Annual ACM Symposium on Principles of Programming Languages, POPL 1989*, pages 329–343, 1989.
22. F. Orejas. Attributed graph constraints. In *Proc. ICGT'08*, volume 5214 of *LNCS*, pages 274–288. Springer, 2008.
23. F. Orejas, H. Ehrig, and U. Prange. A logic of graph constraints. In J. L. Fiadeiro and P. Inverardi, editors, *Fundamental Approaches to Software Engineering, 11th International Conference, FASE 2008*, volume 4961 of *Lecture Notes in Computer Science*, pages 179–198. Springer, 2008.
24. F. Orejas, E. Guerra, J. de Lara, and H. Ehrig. Correctness, completeness and termination of pattern-based model-to-model transformation. In *Submitted*, 2009.
25. K.-H. Pennemann. Resolution-like theorem proving for high-level condition. In *ICGT 08*, volume 5214 of *LNCS*, pages 289–304. Springer, 2008.
26. QVT. `http://www.omg.org/docs/ptc/05-11-01.pdf`, 2005.
27. G. Rangel, L. Lambers, B. König, H. Ehrig, and P. Baldan. Behavior preservation in model refactoring using dpo transformations with borrowed contexts. In *ICGT 08*, volume 5214 of *LNCS*, pages 242–256. Springer, 2008.
28. A. Schürr. Specification of graph translators with triple graph grammars. In *Proc. WG'94*, volume 903 of *LNCS*, pages 151–163. Springer, 1994.