Computations in Graph Rewriting: Inductive Types and Pullbacks in DPO Approach

Maxime Rebout, Louis Féraud, Lionel Marie-Magdeleine, and Sergei Soloviev

IRIT, Université Paul Sabatier
118 Route de Narbonne, F-31062 TOULOUSE CEDEX 9
{rebout,feraud,mariemag,soloviev}@irit.fr

Abstract. In this paper, we give a new formalism for attributed graph rewrites resting on category theory and type theory. Our main goal is to offer a single theoretical foundation that embeds the rewrite of structural parts of graphs and attribute computations which has more expressive power for attribute computations as well.

1 Introduction

Graph grammars have been introduced in the late 1970s [1], then they have been significantly improved up to the 2000s [2]. A lot of significant results are due to H. Ehrig and his colleagues who have conceived an algebraic approach to graph rewriting by the means of category theory [1]. It opened the way to computations with attributes. In this approach, when dealing with model transformations, the transformation process can be viewed as split into two parts: a first one considers the skeleton of the models, *i.e.*, graphs without attributes which can be processed by a double pushout in a graph category, and a second one devoted to computations with attributes. To deal with this part, Ehrig suggests another formalism : the theory of algebraic data types [3]. Our goal when designing the Double Pushout-Pullback approach (abbreviated DPoPb) considered in this paper was on one hand to take advantage of the double pushout approach to implement the rewriting of the structural part of the graphs, and on the other hand to unify in a single formalism (type theory) the attribute computations that occur in graph transformations. Generally, to remain in a unique formalism simplifies the implementations and leads to a more robust software. Moreover, we had in mind to furnish a formalism able to facilitate proofs of properties occurring during transformations such as invariant or pre- or post- conditions preservation. Thus, the main idea of the DPoPb approach is the use of a single formalism for attributed graph rewriting. The power of computations with inductive types is greater due to the presence of functional arguments. The formalism also permits to carry on proofs on transformations.

2 Essentials of the DPoPb

The interest of double pushout (abbreviated DPO) approach is that it offers an algebraic framework based on category theory to perform graph rewriting. The first use of the

pushout to rewrite graphs can be found in Hartmut Ehrig [1]. This approach is now very well known, and we shall give only a sketchy description. In DPO framework, a rewrite rule p is given by three graphs K, L and R and two morphisms $l\colon K \to L$ and $r\colon K \to R$. Intuitively, the elements of L with no pre-image by l will be deleted by the application of the rule and the elements of R with no pre-image by r will be created. The graph K is the "glue" graph linking the graphs L and R. The application of such a rule on a graph G is decomposed in three steps: (i) finding the pattern in G that must be rewritten, *i.e.*, finding an injective graph morphism m from the graph L to the graph G, (ii) deleting elements from the graph G, *i.e.*, computing the pushout-complement of l and m to construct the transition graph D, and (iii) adding new elements, *i.e.*, computing the pushout to find the target graph H.

Later, this theory has evolved. One of the most advanced solutions is given by the DPO in the "adhesive high level replacement categories" (abbreviated HLR approach) [3] in which the graphs can be attributed and typed (not in the sense of type theory). The use of adhesive HLR categories permits to study the (local) CHURCH-ROSSER and parallelism properties, confluence, perform critical pair analysis, *etc.* in graph rewriting systems. In this framework, Σ-algebras are used to encode attributes: information is directly integrated in the graph structure by creating a new "attribute" node for each value of an algebraic sort. In order to attach information to a "structural" node, an edge has to be added between this node and the "attribute" node. The definitions of the rules are very similar to the definitions with simple graphs: some conditions on the morphisms l and r are added in order to ensure that rewriting is possible. A transformation rule $p\colon L \leftarrow K \to R$ is given by three attributed graphs (with variables) K, L, and R and two morphisms $l\colon K \to L$ and $r\colon K \to R$ which have to be injective on the graph structure and isomorphic over the Σ-algebra. In order to describe computations on the attributes, we have to use terms that contain variables; *e.g.*, in the graph R, an attribute $x + y$ can be found if in the graph K the variables x and y are present.

The set of all attributed graphs with the attributed graph morphisms constitutes an adhesive HLR category. Consequently, all the above mentioned properties are present in the framework. The main drawback of this approach is the use of the heterogeneous structures in the definition of an attributed graph (sets and algebraic signatures). The way of dealing with attributes leads to a huge graph: a node is created for each possible value of a variable. Changing the value of an attribute attached to a node consists in canceling the edge connected to the old value to the node and creating a new edge whose target is the new value. Thus "internally" computations should be done by rewriting. If this solution is theoretically acceptable, it is not very efficient and cannot be easily implemented (notice that, in the AGG environment [4,5], the computations of attributes are directly executed in an external programming language (Java)). For instance, the computation of n! needs 2n - 3 steps with three rewrite rules. This way of computations on attributes is artificial. Moreover, it is not possible to express certain classes of recursive functions.

Attributed graphs in the DPOPB approach. In the DPOPB approach, the main idea is to describe in a single formalism the graph structures along with the attributes. Keeping the same conceptual scheme as in the DPO constructions, the goal is to put the

attribute computation to work in a more uniform way by staying within the same theory for implementing computations. The DPOPB solution uses type theory to code the attributed graphs: finite types to describe the structure of the graphs and general inductive types to define the data types. This system is more expressive than the system based on Σ-algebras (inductive types are strictly more expressive than algebraic data types) but the question of its power is not studied in detail in this paper. The precise definition of attributed graphs in the DPOPB approach can be found in [6].

Structure In this paper, the structure of an attributed graph is given by a finite type S^G for the nodes (the numbers of constructors in the inductive type S^G gives the number of the nodes in the graph) and a function $A^G\colon S^G \to S^G \to \mathit{FiniteType}$ which for each pair of nodes associates a finite type describing the edges between the two nodes under consideration.

Attributes For the sake of clarity, only attributes on the nodes will be considered in this paper. Attributes on edges can be processed in a similar way. Conceptually, attaching attributes to the structure requires two steps. First, each node is associated with the types of the attributes we want to bind to that node: this work is realized with a relation between S^G and the set of inductive types representing the data types. Second, the values of the attributes are defined. For each data type attached to a node, two choices are possible: either it is an element of the inductive type representing the data type, or it is undefined, and in this case it will be called a joker and be denoted by ❀. In graph transformations a joker essentially plays the role of variable.

Definition 1. *An attributed graph will be given by (i) a structure (nodes S^G and edges A^G), (ii) a labelling relation to attach the data types to the nodes, and (iii) an attribution function to define a value for each attribute.*

In the complete definition of a DPOPB attributed graph, an equivalence relation is added between the attributed graphs: two graphs will be said equivalent if they differ only by the names of the data types attached to their nodes. To formalize this idea, we use the notion of conform copies of inductive types (see [7,8]). This relation allows to get different names for each data type used to define and then distinguish easily the arguments of the computation function.

Remark 1. Currently we are also working on a generalization to some infinite graphs using the possibility to consider infinite trees as elements of inductive types [9]. The principal idea is that we can decide which part of the graph is considered as strucutre and which part is an attribute [10].

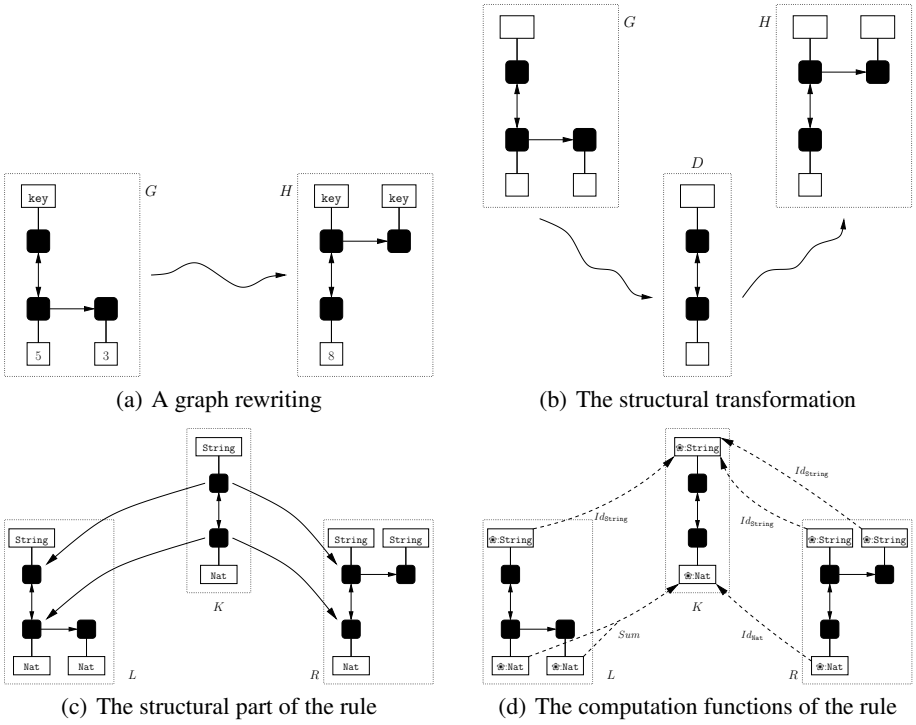
Attributed graph morphisms. The central idea of the DPOPB approach is inspired by the power of the pullback to organize the computation of attribute values. For example, using pullbacks (see, *e.g.*, the introduction of [11]), copies of arbitrary graphs are easily described using only one rule, while in the pushout approach, one rule is needed for each given graph. The use of the pullback (one may see that also as a pushout in a dual category) is justified by the possibility of reversing the arrows of the computation

functions between the attributes: in addition to a classical morphism on the structural part of the graph (sending nodes, respectively edges, of the source graph G to nodes, respectively edges, of the target graph H), functions are added, starting from the data types of H and finishing on the data types of G. In order to do computations with several attributes, these functions may have several data types as arguments. Figures 1(d) and 5 display some examples of computation functions. The image of the values of attributes of the graph H by these functions must then be equal to the values of the attributes of G. The complete morphism construction can be found in [6]. Thus the main construction of DPOPB approach can be seen as using the pushouts in suitable categories but also, for a more intuitive point of view, as a mix between a pushout and a pullback. We have defined the category **AttGraph** whose objects are attributed graphs and whose morphisms are built according to the above requirements. As the structural part of graphs relies on sets (finite types), building the pushout of two morphisms is straightforward. Dealing with the attributes requires some conditions due to the contravariant way how the computation functions work. The restrictions on morphisms are led by the way we want to use these morphisms. They can be defined using two specific morphism classes. The first class is used to describe transformation rules. According to the goals requested in the model transformation, the conditions may vary: for example, on one hand, to be able to undo a transformation, it is needed that no node has been merged (*i.e.* the structural parts of morphisms are injective) and that the attributes have been modified in a reversible way (*i.e.* the computation functions are bijective). One the other hand, complex computations will introduce non bijective functions in the left hand part of the rule. The second class is necessary in application of transformation rules to match exactly a subgraph into a larger graph: the injectivity of the structural morphism and the identities for computation functions are then natural conditions. The restrictions on morphisms are in general satisfied by the morphisms used in graph rewriting when addressing model transformations.

Remark 2. The notion of a metamodel, which is very important in the "Model Driven Architecture", can be translated in our formalism by using a graph to implement the metamodel and a morphism with special condition to depict the relation "to be the metamodel of a model" (*cf.* [12]). Contrary to the HLR approach for metamodels (*cf.* [3]), our system is not limited to one level of metamodeling. The graph transformations with respect to a metamodel are also possible. Moreover, by merging metamodels, exogenous transformations from a metamodel to another one can be implemented (*cf.* [12]).

The importance of reversing arrows. Let us study a simple example. Figure 1(a) displays the transformation to be applied to a source graph: the two numbers (attributes of the graph G) have to be added and the result has to be stored in an attribute of H; the node carrying the second attribute is deleted during the transformation. In parallel, a new node is created on H with a copy of the string ("`key`") from G.

If we just look at the structure of the graph, it is easy to guess the structural part of the context graph D (see figure 1(b)), *i.e.*, the result of the first pushout-complement. Since we do not merge nodes during this transformation, the structure of this graph is equal to $G \cap H$. As a node is deleted from G during this transformation, the graph D has only two nodes and a bidirectional edge. Because every data type of the context



(a) A graph rewriting

(b) The structural transformation

(c) The structural part of the rule

(d) The computation functions of the rule

Fig. 1. Reversing the arrows.

graph will also be in G and H, the choice for the data types attached to the two nodes of D is very limited. Since there are only two attributes in common between G and H, D cannot possess more than two attributes and if we want to conserve information during the transformation, D must have a natural number and a character string as attributes. Writing the structural morphism for this transformation is straightforward. Figure 1(c) shows the structural part of the morphism l (respectively r) between the context graph K of the rule and the left-hand side L (respectively the right-hand side R). On this diagram, only the arrows encoding the image of the nodes of the graph K are represented. It is now possible to see why reversing the orientation of the computation function is natural[1]. In the morphism l, the two natural numbers from L have to be added and the result must be stored in the attribute of K. Thus, intuitively speaking, starting from L to work with its attributes before we put the result in K is more natural. Moreover, since the sum is not injective, it is impossible to find an inverse for it. Consequently, it is then much more relevant to inverse the orientation of arrows for the sum function. For the same reason, on the right-hand side of the transformation, copying the string

[1] This problem was less visible in the approach where the real computation was done by an external mechanism.

suggests also to reverse the arrows[2]. In the classical approach, the value of an attribute can only be sent to one place; but by changing the orientation of the arrows for the functions, the attribute in graph R can easily "go and pick up" any value of attributes in the graph K; thanks to this mechanism, several attributes can then share the same value. The computation functions for the considered rule are given in Figure 1(d).

The above considerations lead us to define graph morphisms with forwards arrows for the structural part - the usual way to define pushouts - and with backwards arrows for the attribute computations - as in pullbacks. This is why we have called our transforming approach "Double Pushout Pullback".

The importance of inductive types. To emphasize the importance of using inductive types in our approach, we will give an example. In the HLR approach, with the use of algebraic signature to encode attribute values, the computation of $n!$ needs three transformations rules (*cf.* [4]): (i) the first one creates a sequence of nodes with attributes decreasing from n to 2, (ii) the second rule ensures that this sequence is over, (iii) the third rule multiplies the last two numbers of the sequence and deletes the last node. Computing $n!$ requires then $2n-3$ steps. With inductive types, defining recursive functions is natural. Consequently, in the DPOPB approach, the computation of $n!$ needs only one rule and one step of attribute graph rewriting. Of course, computation of recursive functions requires many steps but it is included in a standard framework which is part of the system based on inductive types, and so, a natural part of our formalism.

3 The DPOPB Graph Rewriting System

Applying transformation rules to a source graph according to the DPOPB approach The classes $\mathcal{M}$ and $\mathcal{N}$ are the classes of morphisms mentioned in section 2. We recall that this classes ensure the construction of rules and their applications. Their precise definitions can be found in [12].

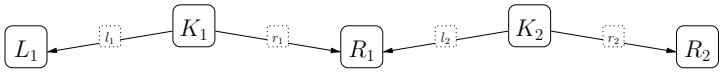
Definition 2. *A transformation rule p (or a production) is given by three attributed graphs K, L and R with two morphisms $l\colon K \to L$ and $r\colon K \to R$ belonging to the class $\mathcal{M}$. Moreover, it is required that only jokers can be sent to jokers in these morphisms.*

Definition 3. *Let $p\colon (L \leftarrow K \rightarrow R)$ a transformation rule and G an attributed graph. A match is a morphism $m\colon L \to G$ belonging to the class $\mathcal{N}$.*

The graph K will be called the interface of the rule, L the left-hand side and R the right-hand side. Let $p\colon (L \leftarrow K \rightarrow R)$ be a transformation rule, G an attributed graph and m a match. The application of the rule p to the graph G is similar to the one introduced with the DPO approach [1]: if the gluing condition is fulfilled then the pushout-complement of l and m is computed and then the pushout of the right-hand side is constructed. The target attributed graph H is obtained.

[2] This example shows that right hand side cannot carry computation information but only copying mechanism. It is a slight disavantage shared with other known approaches. It can be compensated by combination of several rules.

In some cases, the process of rule application may be more complex. Let $p_1 : (L_1 \leftarrow K_1 \rightarrow R_1)$ and $p_2 : (L_2 \leftarrow K_2 \rightarrow R_2)$ be two rules such that $L_2 = R_1$, it is then possible to "concatenate" p_1 and p_2 in a single composed rule; for instance, the first part creates a place holder and the second one puts the attribute value in it.



In difference from the HLR approach, the factorization of a composed rule into a single production is not always possible because the pullback of morphisms r_1 and l_2 does not always keep all the useful information for the transformation[3]. Examples of composed rules are given in section 4. During the application of such a rule, the morphism m^* computed after the first double-pushout is used as the match for the second part of the rule.

Application conditions Sometimes, it is useful to better control the application of the rules. For example, it could be forbidden to apply a rule if some conditions involving nodes or attributes hold. For this, some application conditions are used (see [13]). A positive application condition (PAC) means that some context in the graph G is required and a negative application condition (NAC) means that some context is forbidden. An application condition is given by an additional graph N and a morphism between the left-hand side of the rule and this new graph. A rule r can be qualified by several application conditions and the transformation of a graph G by r and a match m will be possible only if all the application conditions hold. Thanks to NACs, it is possible to test the presence and the absence of an element (node, edge or data type) in the graph, to forbid the application of a rule if an attribute is equal to some value. The detailed definition of NAC can be found in [12].

Confluence and termination.

Definition 4. *Let $p_1 = L_1 \leftarrow K_1 \rightarrow R_1$ and $p_2 = L_2 \leftarrow K_2 \rightarrow R_2$ be two production rules. Two direct derivations $G \overset{p,m}{\Longrightarrow} H$ and $G \overset{p',m'}{\Longrightarrow} H'$ are parallelly independent if there exist morphisms $k_1 : L_2 \rightarrow D_1$ and $k_2 : L_1 \rightarrow D_2$ in the class $\mathcal{N}$ such that (cf. Fig. 2):(i) $l_2^* \circ k_2 = m_1$ and $l_1^* \circ k_1 = m_2$, (ii) morphisms $r_2^* \circ k_2$ and $r_1^* \circ k_1$ belong to the class $\mathcal{N}$.*

Definition 5. *Let $p_1 = L_1 \leftarrow K_1 \rightarrow R_1$ and $p_2 = L_2 \leftarrow K_2 \rightarrow R_2$ be two production rules. Two derivations $G \overset{p_1,m_1}{\Longrightarrow} H_1$ and $H_1 \overset{p_2,m_2}{\Longrightarrow} G'$ are sequentially independent if there exist morphisms $k_2 : R_1 \rightarrow D_2$ and $k_1 : L_2 \rightarrow D_1$ in the class $\mathcal{N}$ such that (cf. Fig. 2):(i) $l_2^* \circ k_2 = m_1^*$ and $r_1^* \circ k_1 = m_2$, (ii) morphisms $l_1^* \circ k_1$ and $r_2^* \circ k_2$ belong to the class $\mathcal{N}$.*

[3] It may seem that one does not need composed rules in HLR (they can be replaced by one rule). The simple examples involving recursion where the number of applications varies show it is not really the case.

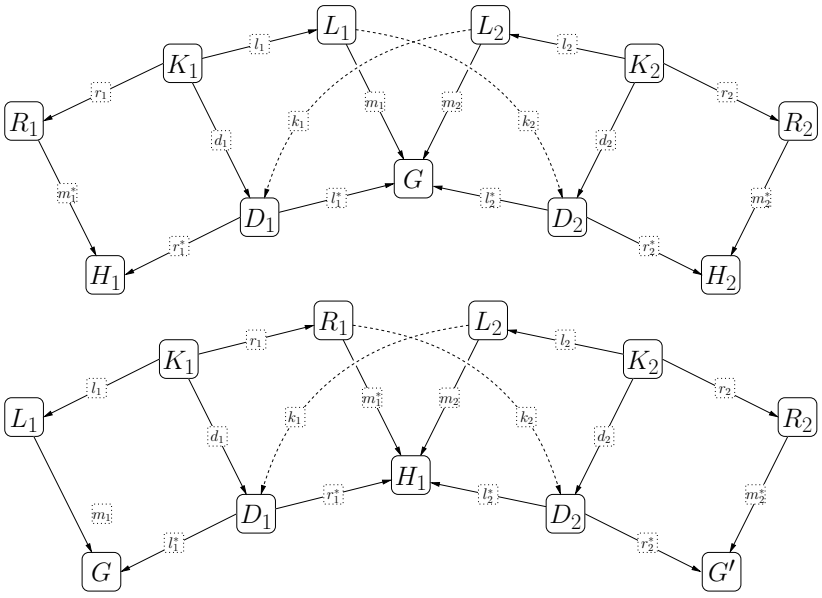


Fig. 2. Parallel and sequential independence.

The parallel and sequential independences provide local confluence. Otherwise, the notion of critical pairs and strict confluence are used [3,12]. Then, the well-known theorem is still true in our formalism: a transformation system is locally confluent if every critical pair is strictly confluent. We have also developed some criteria for termination based on three types of layers: (i) creation layer, (ii) computation layer and (iii) deletion layer (concerning creation and deletion layers, see [14]; the computation layer has been introduced in [12]). This approach was sufficient to ensure confluence and termination in standard systems of model transformations.

4 Putting the Transformations to Work

From class diagram to relational data base. The requirements for this transformation are presented in [15]. We give here just a basic variant where primary key, class inheritance and non-persistent class are not taken into account (see [16]). The "type graph" for the translation is the merging of the two abstract graphs describing respectively the metamodels of UML class diagrams and relational data bases along with connections by edges describing the correspondences between the concepts of source and target language (see [6]).After the translation, the source graph and these links are removed; this final step is omitted in this case study.

The translation is decomposed into four steps representing by four rules: (i) for each class, a table is created, (ii) for each primary attribute, a column is created (these rules are depicted in [6]), (iii) for each attribute with a class as attribute type, a column and a

foreign key is created (see figure 3) and (iv) for each association, a column and a foreign key are created (computation functions are similar to the ones involved in the third rule).

The originality of the attribute computation process is stressed by exhibiting on the rules the computation functions involved in non-trivial manipulations. For all rules, there are negative application conditions: in order to avoid to apply the same rule for the same match, the right part of each rule is also a NAC.

Collecting the ancestors. One may consider an UML class depicting a person. It possesses the attributes "name" and "ancestors" depicting respectively the name of the person and the list of his ancestors. A person is connected to his parents using the relation "parents". Gathering the list of parents in a class requires the computation of the transitive closure of the "parents" relation. In the transformation language ATL [17], this computation needs a recursive function implemented in a helper.

```
helper context Base!Person
def:getAncestors:Sequence(Base!Person)=
self.parents->union(self.parents
->collect(e|e.getAncestors))->flatten();
```

This recursive computation can be implemented in the DPOPB approach thanks to four rewrite rules avoiding recursivity (see figures 4 to 7): (i) the first rule is necessary to initialize the collection by creating the list as an attribute on the initial node. A negative application condition comes along in order to prevent to iterate its application, (ii) the second rule is used to start climb up the ancestor tree. A temporary edge `Temp` is created to keep track of the computation, (iii) the third rule will follow the path in the graph until the last ancestor. The list is growing at each step and the temporary edge is moved and (iv) the last rule allows to bring back the list of ancestors to the start node.

The termination criteria mentioned in section 3 fit correctly this example. Indeed, by sorting the four rules into layers: (i) the first rule is placed in a creation layer, (ii) a computation layer is composed by the second and the third rules and (iii) the fourth rule defines a deletion layer. It is easy to see that each layer terminates (for example, the creation layer terminates thanks to the negative application condition). Hence, this transformation system will always end. The previous example does not take yet full advantage of the power of inductive types. In contrast with the HLR approach, constructing attributed graphs with inductive types permits to process potentially infinite graphs (recently we considered rewriting with infinite trees as attributes). Thanks to them, it will be possible to address model transformations where infinite models are required such as in observation of dynamic behaviour of a system, continuous stream of data generated by an intelligent sensor, complex event processing, limit constructions, *etc*. Thus DPOPB approach may open a new track to areas of modeling not currently addressed for lack of adequate formalisms.

5 Discussion and Conclusion

The interest of the method presented in this paper rests on the expression in a single formalism of graph structure rewrites and attribute computations that also rely on category

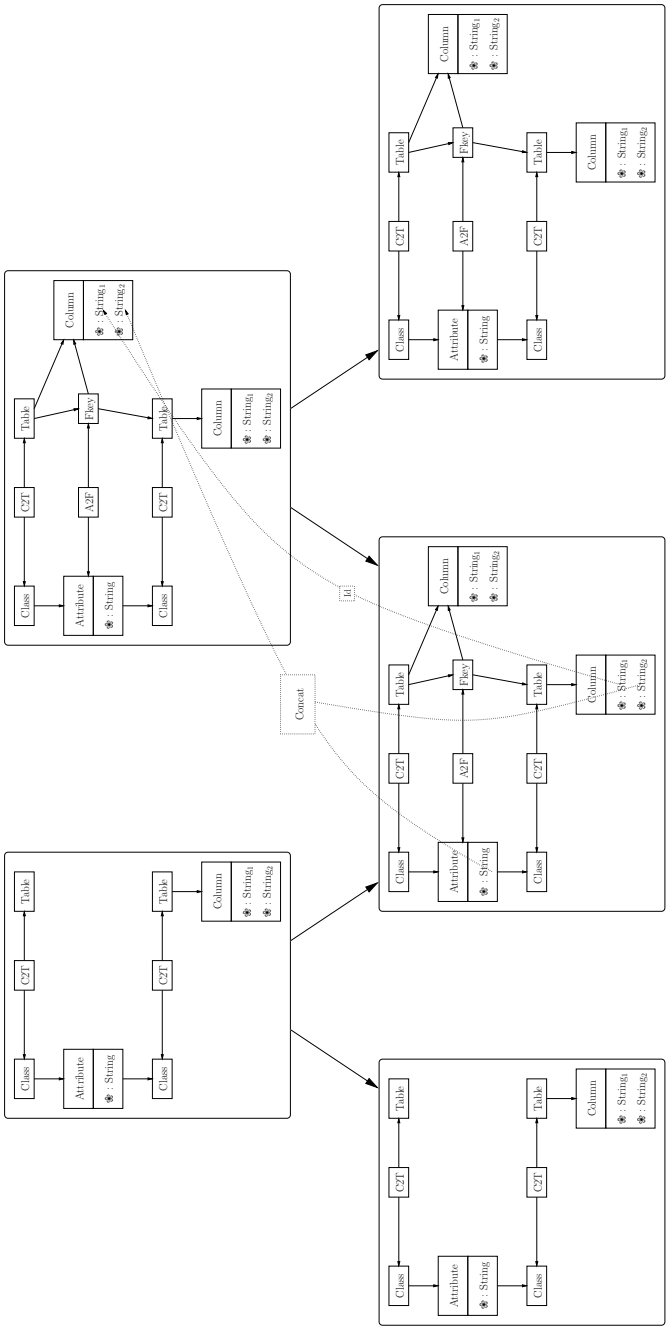


Fig. 3. Rule 3. A complex rule to compute the foreign key: the first part creates the placeholder for the attribute and the second part add the value computed with the "Concat" function to this place.

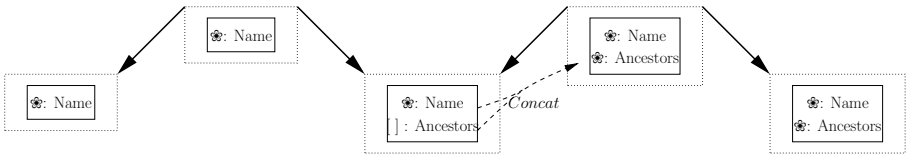


Fig. 4. Rule 1. Initialization.

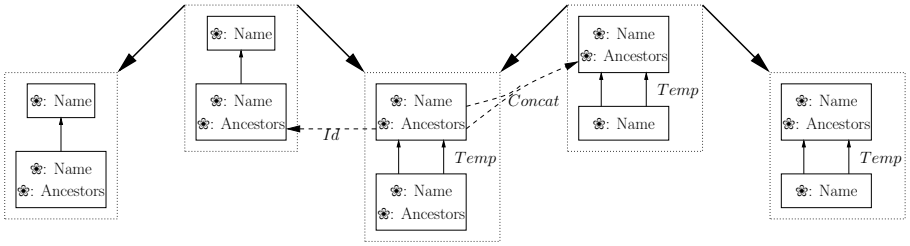


Fig. 5. Rule 2. Collecting the first ancestor.

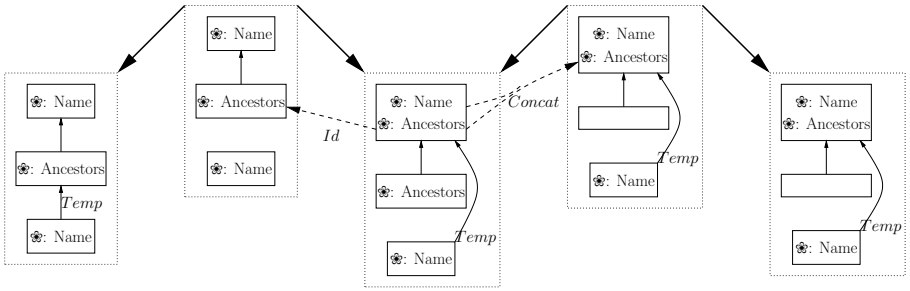


Fig. 6. Rule 3. Following the path to collect all the ancestors.

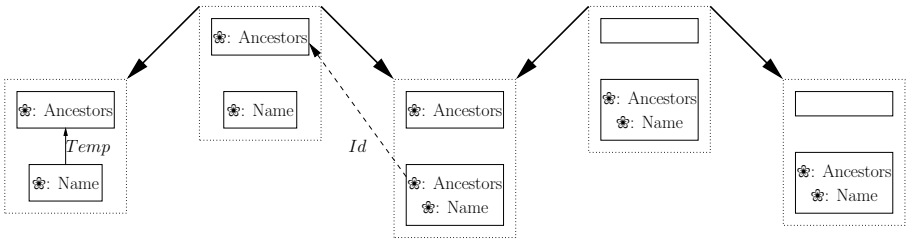


Fig. 7. Rule 4. Completion.

theory. The DPOPB differs from the standard DPO graph rewrite [3] due to its original way to build graph morphisms: the attribute arrows are reversed as in a pullback operation.

The comparison of the DPOPB approach with other existing formalisms shows that this formalism possesses the main characteristics of similar ones which are the background of AGG, PROGRES, VIATRA, ATOM^3. As DPOPB relies on inductive types, proofs of transformation properties should be facilitated. Consequently, the study of transformation verification is a first application of our work. As it has been stated before, an implementation of DPOPB rewriting is currently under way. This implementation is based on the functional language Haskell. The Coq proof assistant whose libraries contain categorical tools will be also used in connection with proofs considerations. Of course, the performances of such a system will have to be discussed with respect to other existing graph rewriting environments. Similarly to the graph rewriting formalisms mentioned above, our approach can be useful to adress model and metamodel transformations. Let us summarize some of its more standard features according to certain criteria defined in [16].

Typing. As in [18], the concept of metamodel is defined by the means of "type graphs". Thus the conformity of a model (graph) with respect to a metamodel (type graph) is expressed by a graph homomorphism. Thus, the "conformity relation", sometimes called "instantiation" remains in the theoretical framework. Moreover, successive typing is possible in the DPOPB approach.

Precondition. The left-hand side of rules constitutes the main precondition for applying them. Some additional conditions such as positive or negative application condition can be added. They are defined according to the category theory in terms of graphs, morphisms and pushouts.

Postcondition. In some cases, such as in the example of the transformation depicted in section 4, the right-hand side of rule is the left-hand side of another rule constituting the so-called composed rules. Such rules avoid the duplication left-hand sides or right-hand sides and are useful to implement complex transformation steps. In certain approaches, composed rules can be replaced by one rule, but this possibility is limited in case of recursive computations (*cf.* 3).

Actions. The construction of the structural part of the target graph using double pushout is similar to the one offered by AGG [4] for instance. The attribute computations are obtained by categorical calculations. These computations can be implemented in Coq or in a functional language like Haskell.

Control. Currently, the DPOPB approach supports non-determinism for rule application and for match selection within the source graph. Solutions involving priorities [19] or layers [14] can be used.

Correctness. Thanks to its theoretical foundation using inductive types in a dependent type framework, the DPOPB approach offers a suitable environment to consider correctness. Despite the fact that the **AttGraph** category does not belong to adhesive HLR categories, the termination, local CHURCH-ROSSER, and critical pair analysis have been proved in standard cases (see [12]).

Complexity. In general, the complexity of graph rewriting can be huge. This complexity is not specific to our approach: already the match of a subgraph into a larger

graph is a NP-complete problem. Furthermore, for specific use (for example the verification of a transformation or the rewrite of a graph with a small set of rules, *cf.* [12]), the complexity is acceptable and our system does not increase the computation time in comparaison to other approaches.

Implementation. An implementation of the system is currently under way. The use of non-standard reductions in type theory, especially for finite types, will contribute to simplify the computation of attributes (*cf.* [8]). The core of the system is developed using Haskell: this functional language is well-suited in order to implement categorical constructions [20]. The verification of computations will be processed using Coq.

Some of our results mentioned above were published in [6]. Main new results of this paper are the tools developped to prove confluence and termination of a model transformation system. We currently work on the generalization of our approach for infinite graphs. Inductive types seem a natural way to include the graphs with inductively defined sets of nodes in our approach. We presented also several examples easily generalizable in this direction.

Acknowledgments. We thank Reiko Heckel, Konstantin Verchinine and Ralph Sobek for helpful discussions and comments. This work was supported by the European TYPES project and the pluri-disciplinary PEPS action from French CNRS.

References

1. Ehrig, H.: Introduction to the Algebraic Theory of Graph Grammars (a Survey). In: Claus, V., Ehrig, H., Rozenberg, G. (eds.) Graph-Grammars and Their Application to Computer Science and Biology. LNCS, vol. 73, pp. 1–69. Springer (1978)
2. Rozenberg, G. (ed.): Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations. In: Rozenberg, G. (ed.) Handbook of Graph Grammars, World Scientific (1997)
3. Ehrig, H., Prange, U., Taentzer, G.: Fundamental Theory for Typed Attributed Graph Transformation. In: Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G. (eds.) ICGT. LNCS, vol. 3256, pp. 161–177. Springer, Heidelberg (2004)
4. AGG Homepage, http://tfs.cs.tu-berlin.de/agg
5. Wolz, D.: Colimit Library for Graph Transformations and Algebraic Development Techniques (1998)
6. Rebout, M., Féraud, L., Soloviev, S.: A Unified Categorical Approach for Attributed Graph Rewriting. In: Hirsch, E., Razborov, A., Semenov, A., Slissenko, A. (eds.) CSR. LNCS, vol. 5010, pp. 398–409. Springer, Heidelberg (2008)
7. Chemouil, D.: Isomorphisms of Simple Inductive Types Through Extensional Rewriting. Math. Structures in Computer Science 15(5) (2005)
8. Chemouil, D., Soloviev, S.: Remarks on Isomorphisms Of Simple Inductive Types . In: Mathematics, Logic and Computation , Eindhoven, 04/07/03-05/07/03, ENTCS 85, 7, pp. 1–19. Elsevier (2003)
9. Jouault, F., Bézivin, J., Barbero, M.: Towards an Advanced Model-driven Engineering Toolbox. Innovations in Systems and Software Engineering 5(1), 5–12 (2009)
10. Wyk, E.V., Moor, O.d., Backhouse, K., Kwiatkowski, P.: Forwarding in Attribute Grammars for Modular Language Design. In: CC '02: Proc. of the 11th International Conference on Compiler Construction, pp. 128–142. Springer-Verlag (2002)
11. Kahl, W.: A Relational-algebraic Approach to Graph Structure Transformation. PhD thesis, Universität der Bundeswehr München (2001)

12. Rebout, M.: Une approche catégorique unifiée pour la récriture de graphes attribués. PhD thesis, Université Paul Sabatier (2008)
13. Habel, A., Heckel, R., Taentzer, G.: Graph Grammars with Negative Application Conditions. Fundamenta Informaticae 26(3/4), 287–313 (1996)
14. Ehrig, H., Ehrig, K., Taentzer, G., de Lara, J., Varró, D., Varró-Gyapay, S.: Termination Criteria for Model Transformation. In: Cordy, J.R., Lämmel, R., Winter, A. (eds.) Transformation Techniques in Software Engineering. Dagstuhl Seminar Proceedings, vol. 05161, Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany (2005)
15. Bézivin, J., Rumpe, B., Schürr, A., Tratt, L.: Model Transformations in Practice Workshop. In: Bruel, J.M. (ed.) MoDELS Satellite Events. LNCS, vol. 3844, pp. 120–127. Springer, Heidelberg (2005)
16. Ehrig, K., Guerra, E., de Lara, J., Lengyel, L., Levendovszky, T., Prange, U., Taentzer, G., Varró, D., Varró-Gyapay, S.: Model Transformation by Graph Transformation: A Comparative Study. In: MTiP 2005, International Workshop on Model Transformations in Practice (Satellite Event of MoDELS 2005) (2005)
17. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: Atl: A Model Transformation Tool. Science of Computer Programming 72(1-2), 31–39 (2008)
18. Heckel, R.: Graph Transformation in a Nutshell. In: Bézivin, J., Heckel, R. (eds.) Language Engineering for Model-Driven Software Development. Dagstuhl Seminar Proceedings, vol. 04101, Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany (2004)
19. de Lara, J., Vangheluwe, H.: Atom^3: A Tool for Multi-formalism and Meta-modelling. In: Kutsche, R.D., Weber, H. (eds.) LNCS, vol. 2306, pp. 174–188. Springer, Heidelberg (2002)
20. Schneider, H.J.: Implementing the Categorical Approach to Graph Transformations With Haskell. In: An Introduction to the Categorical Approach (Draft March 7, 2007)