Electronic Communications of the EASST
Volume 6 (2007)

Proceedings of the Sixth International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2007)

Adding Recursion to Graph Transformation.

Esther Guerra, Juan de Lara

14 pages

Guest Editors: Karsten Ehrig, Holger Giese
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Adding Recursion to Graph Transformation.

Esther Guerra[1], **Juan de Lara**[2]

[1] eguerra@inf.uc3m.es
Dep. Ingeniería Informática
Universidad Carlos III de Madrid (Spain)
[2] jdelara@uam.es
Escuela Politécnica Superior
Universidad Autónoma de Madrid (Spain)

Abstract: In this paper we define recursive rules in the double pushout approach (DPO) to graph transformation. Classical DPO rules are extended with a *base case condition* and a *recursion condition*. Mechanisms are provided to pass the match from both conditions to the rule's left hand side, and also between two consecutive steps in the recursion. The approach is useful when recursive structures (such as inheritance hierarchies, nested component hierarchies, networks of functional blocks, etc.) have to be processed. Although we present the recursion for DPO, it can also be adapted to other approaches to graph and model transformation. We present examples for model transformation, model simulation and model optimization in different application domains.

Keywords: Graph Transformation, Double Pushout, Recursion.

1 Introduction

Graph transformation [Roz97] is becoming increasingly popular to express computations on graphs due to its formal, declarative and graphical nature. One of the most popular formalizations of graph transformation is the double pushout approach (DPO) [EEPT06], which uses category theory to model rules and derivations. Graph transformation has been used in many application areas, such as modelling with visual languages [Min02], visual simulation [LV04], model transformation [EGL+05] and refactoring [EJ04]. The manipulation of structures with nested, iterated or recursive elements is common in many of these areas.

The formal nature of DPO graph transformation allows interesting analysis techniques, for example to investigate confluence, termination and rule independence [EEPT06]. Moreover, the categorical framework has lifted the results from graphs to any (weak) adhesive HLR category [LS04, EEPT06] (short AHLR category). However, compared to other approaches [BV06, KASS03, SWZ99, NNZ00], it lacks expressivity when handling complex structures involving recursion, iteration or nesting. Processing such structures usually implies performing a certain action in their different parts (e.g. copying all the attributes of a class to its children). Moreover, sometimes the structure has to be traversed in a certain order (e.g. when propagating a change in a network of logical gates). Having high-level constructs to process these kinds of structures is interesting for model transformation, but as we show, it is also useful for other manipulations such as optimization and simulation.

Usually, there are two options when processing a recursive structure with DPO rules. As each element has to be consecutively processed, a solution is encoding helper control elements in the graph, which guide the application of the rules. However, this is sometimes undesirable or impossible, as it implies modifying the meta-model (or type graph) of the model to be transformed. Another possibility is to *flatten* the structure. For example, performing the transitive closure (by adding *ancestor* edges) can flatten an inheritance hierarchy. Again, this solution implies a modification of the type graph as well as pre- and post- processing phases.

In this paper we propose *recursive rules*, which enhance the expressive power of DPO rules in order to make the recursive processing of structures easier. We extend classical DPO rules with mechanisms for passing matching information between consecutive rule applications, and to guide rule execution by traversing structures recursively, where the rule's action is executed at each step in the recursion. We present our approach in the AHLR framework, in such a way that it becomes valid for any AHLR category. In particular, we show an instantiation for attributed typed graphs. Our approach has several benefits. On the one hand, there is no need to add extra elements to process the recursive structure. This usually leads to simpler and higher-level rules as we abstract from "accidental details", concentrating on the essence of the problem. On the other hand, the execution of a recursive rule can be more efficient than several DPO simple rules, as the matches are guided through the structure. Moreover, the framework can be adapted to other graph and model-transformation approaches.

Paper organization. Section 2 gives an overview of transformation formalisms that consider the processing of recursive structures. Section 3 introduces the DPO approach. Section 4 presents our proposal for recursive rules. Section 5 shows additional examples for model simulation and transformation. Finally, section 6 ends with the conclusions and future work.

2 Related Work

Some approaches to model transformation are found in the literature to handle recursive applications of a rule. Usually, they are based on the use of some control flow language that guides the transformation execution. For example, GReAT [KASS03] provides hierarchical data flow diagrams for control execution, where rules can be placed on blocks and participate in recursive calls. Control flow in VIATRA [BV06] is specified by means of Abstract State Machine (ASM) programs, while in Fujaba [NNZ00] it is done by means of story diagrams. In the OMG's QVT [OMG] specification, complex transformations can be implemented by using the Operational Mapping Language either in an imperative or in an hybrid approach.

Other model transformation languages embed control mechanisms in the rules. One example is UMLX [Wil03], which provides rule encapsulation, thus allowing composition and some degree of recursion. However, traversing a structure can be incomplete if a match of the rule is not found in some recursion step. MOLA [KBC05] incorporates mainly loopings in the rules as graphical control structure, and allows the transitive closure and the traversal of the recursive structure in a single rule. However, the recursion semantics is not formally defined.

In addition to control languages, some approaches offer recursive mechanisms in the left hand side (LHS) of the rules. These are conditions that can be recursively evaluated by traversing certain structures in the graph. For example, PROGRES [SWZ99] and Fujaba provide path

expressions. Thus, given an initial match, edges can be matched by testing their existence or by traversing them. Another example is VIATRA, where queries on graphs can be expressed by generalized (recursive) graph patterns, which may contain a nesting of positive and negative patterns of arbitrary depth. However, neither path expressions nor recursive patterns can guide the rule execution. That is, they are expanded by a recursive evaluation, but the rule that contains them is applied once. Consecutive rule applications require the use of a control language and a mechanism to pass the elements matched in previous executions to the next execution step.

With respect to approaches based on DPO, there are some attempts to increase the expressiveness of DPO rules for refactoring [EJ04]. However, these do not consider recursive application of rules. In parallel graph transformation [Tae96], standard DPO rules are extended by allowing certain parts of the rules to be instantiated an arbitrary number of times, and providing synchronization mechanisms controlling the way the matching should occur. Thus, the approach is a way to describe in a concise way a (possibly infinite) number of rules in which a certain part is replicated. Again, this is a mechanism for rule expansion, where the rule is applied once.

3 The Double Pushout Approach

In this section we give a brief overview of the double pushout approach (DPO) to graph transformation. See [EEPT06, Roz97] for a more extensive presentation.

Graph grammars are made of rules with a left and right hand side (LHS and RHS). When a rule is applied to a graph G (the *host graph*), an occurrence of the LHS (a matching morphism) has to be found in G, which can be then substituted by the rule's RHS. DPO uses category theory to model rules and derivations, and its theory has been lifted from graphs to (weak) AHLR categories [LS04, EEPT06]. These categories are based on a distinguished class $\mathscr{M}$ of monomorphisms. Examples of AHLR categories are graphs, typed graphs, P/T nets (indeed a weak AHLR category) and attributed typed graphs. Thus, not only graphs, but also objects in any (weak) AHLR category $(\mathbf{C}, \mathscr{M})$ can be rewritten using DPO rules.

DPO rules are modelled using three components: L, K and R. L contains the necessary elements to be found in the object to which the rule is applied. K (the gluing object) contains the elements that are preserved and R those that should replace the identified part in the object being rewritten. Roughly, $L-K$ are the elements that should be deleted by the rule application, while $R-K$ are the elements that should be added. We present these two concepts in the following definitions, taken from [EEPT06].

Definition 1 *(DPO rule) Given a (weak) AHLR category* $(\mathbf{C}, \mathscr{M})$*, a DPO* rule $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ *consists of three objects* L*,* K *and* R *called left hand side, gluing object and right hand side respectively, and morphisms* $l : K \to L$*,* $r : K \to R$ *with* $l, r \in \mathscr{M}$*.*

Definition 2 *(DPO derivation) Given a DPO rule* $p = (L \xleftarrow{l} K \xrightarrow{r} R)$*, an object* G*, and a morphism* $m : L \to G$ *called match. A* direct derivation $G \overset{p,m}{\Longrightarrow} H$ *from* G *to an object* H *is given by the diagram to the left of Fig. 1, where (1) and (2) are pushouts. A sequence* $G_0 \Rightarrow G_1 \Rightarrow \ldots \Rightarrow G_n$ *of direct derivations is called a* derivation *and is denoted as* $G_0 \overset{*}{\Rightarrow} G_n$*.*

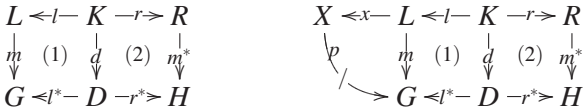


Figure 1: DPO Direct Derivation (left). Direct Derivation by DPO Rule with NAC (right).

Sometimes, rules are equipped with application conditions [EEPT06] constraining their applicability. For simplicity, we only deal with negative application conditions (NACs). These have the form $NAC(x)$, where $x\colon L \to X$ is a morphism. Morphism $m\colon L \to G$ satisfies $NAC(x)$ if there is no morphism $p\colon X \to G$ in $\mathscr{M}'$ with $p \circ x = m$ (see right of Fig. 1). $\mathscr{M}'$ is an additional class of distinguished morphisms than can be $\mathscr{M}$ if the latter is the class of all monomorphisms [EEPT06]. Next, we define the concepts of AHLR system, grammar and language.

Definition 3 *(AHLR system, grammar and language) An* AHLR system $AHS = (\mathbf{C}, \mathscr{M}, P)$ *consists of a (weak) AHLR category* ($\mathbf{C}$, $\mathscr{M}$) *and a set of productions* P. *An* AHLR grammar $AHG = (AHS, S)$ *is an AHLR system together with a distinguished start object* S. *The* language L *of an AHLR grammar is defined by* $L = \{G \mid \exists S \overset{*}{\Rightarrow} G\}$.

Example and Motivation. Fig. 2 shows some DPO rules in the category of typed graphs $\mathbf{Graph_{TG}}$, where objects are tuples $(G, type_G)$. The first element is a graph, and the second one a typing function $type_G\colon G \to TG$ from G to a distinguished graph called the type graph. The type graph in the example is taken from a Role Based Access Control system. It models hierarchies of roles (through relation *parent*), which can be granted permission (relation *perm*) to execute certain functions. The rules present together in a single graph the L, K, R and X components. Elements in $L - K$ are marked as "del", elements in $R - K$ as "new" and elements in $X - L$ as "NAC". We follow a UML-like notation, where the types are shown after a colon.

Rules in the example are used to eliminate redundant permissions in role hierarchies. A permission is redundant for a role if an ancestor already defines it. The rules first calculate the transitive closure of relation parent by adding helper edges of type *anc*. Rule *createDirectAncestor* creates such edge to a direct parent. The iterated execution of rule *createAncestor* performs the transitive closure. Rule *removeRedundantPermission* removes relation *perm* from a role if an ancestor already has the same permission. We use an execution control structure for rules based on layers. The three previous rules are assigned the layer one and are applied as long as possible. Once no rule in this layer can be applied, the next layer is executed. The second layer contains rule *deleteAncestor*, which deletes the helper *anc* edges (i.e. a post-processing step).

Note how, in order to detect redundant permissions, we need to add *anc* helper edges from each role to all its ancestors. This is done because we do not know how long is the path of parent edges starting from a given role. Without the helper edges one could build different rules (similar to *removeRedundantPermission*) to eliminate the redundant permission when there is a direct connection between two roles, when they are separated by a path of two, of three and so forth. However, for arbitrary structures, this does not work as we may need arbitrarily many rules. In addition, DPO simple rules only have information of nodes and edges matched by its LHS. There is no control mechanism that allows moving through a given structure and pass the

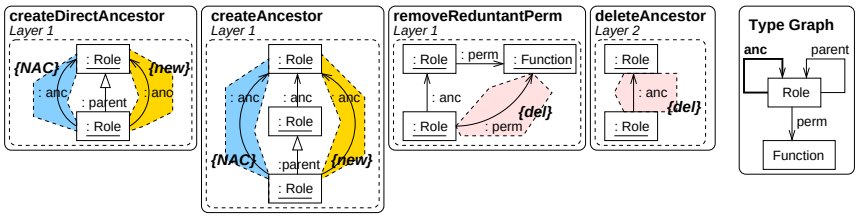


Figure 2: Example DPO Rules.

matching information between consecutive derivations. Hence, control information has to be encoded in the data or, as in this case, the recursive structure has to be flattened.

The solution of adding helper edges (*anc* in the example) is not optimal. First, we have to modify the type graph (see Fig. 2). This is undesirable or even impossible in real applications, if the type graph is a standard meta-model of some modelling language (such as UML), and is being used by other users and tools. Moreover, if several computations have to be performed, it is not feasible that each one of them adds different helper structures to the type graph. Second, there is a pre-processing phase in which helper edges are explicitly added, and a post-processing phase in which the edges have to be removed. These two phases produce a computation overload. In the next section we propose a solution to alleviate these problems.

4 DPO Recursive Rules

In this section we extend DPO rules with several artefacts to model *conditions* for the base and recursive cases, as well as a mechanism to pass part of the match between the base and recursive cases, and between two consecutive steps in the recursion.

Definition 4 *(DPO recursive rule) A DPO recursive rule* $p_r = (L \xleftarrow{l} K \xrightarrow{r} R, I^b, I^r, (I^j \xleftarrow{i^j} P^j \xrightarrow{p^j} L)_{j\in\{b,r\}}, (I^j \xleftarrow{i^{jr}} P^{jr} \xrightarrow{p^{jr}} I^r)_{j\in\{b,r\}})$ *is made of:*

- *A DPO rule* $L \xleftarrow{l} K \xrightarrow{r} R$*, a base condition* I^b *and a recursion condition* I^r*.*
- *The relations between the base* $(j = b)$ *and recursion* $(j = r)$ *conditions and* L *by means of their common elements (object* P^j*),* $(I^j \xleftarrow{i^j} P^j \xrightarrow{p^j} L)_{j\in\{b,r\}}$*, with* $i^j, p^j \in \mathscr{M}$*.*
- *The relation between the base and the recursion conditions by means of their common elements (object* P^{br}*),* $I^b \xleftarrow{i^{br}} P^{br} \xrightarrow{p^{br}} I^r$*, with* $i^{br}, p^{br} \in \mathscr{M}$*.*
- *The relation between the recursion condition for two consecutive steps in the recursion by means of their common elements (object* P^{rr}*),* $I^r \xleftarrow{i^{rr}} P^{rr} \xrightarrow{p^{rr}} I^r$*, with* $i^{rr}, p^{rr} \in \mathscr{M}$*.*

with the constraint that P^b *and* P^r *have to be preserved by the DPO rule application, that is, there are morphisms* $a\colon P^b \to K$*,* $b\colon P^r \to K$ *s.t.* $l \circ a = p^b$ *and* $l \circ b = p^r$*, i.e. triangles (1) and (2) in Fig. 3 commute.*

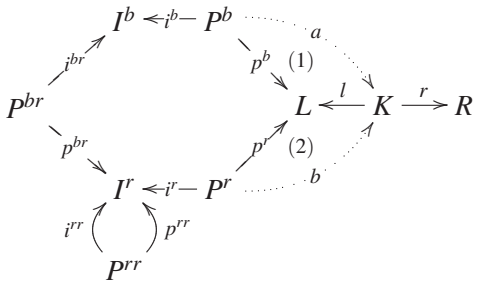
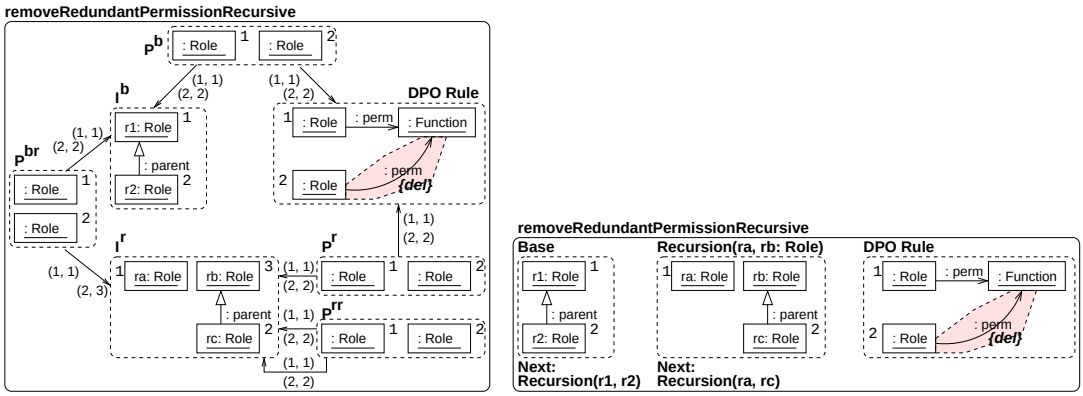


Figure 3: Formalization of a DPO Recursive Rule.

Fig. 4 shows a recursive rule that eliminates redundant permissions in role hierarchies, equivalent to the set of standard DPO rules in Fig. 2. To the left, the rule is shown according to the theory, to the right using a more compact and intuitive notation that will be used throughout the paper.



(a) (b)

Figure 4: A DPO Recursive Rule Example (a) Theoretical Notation. (b) Compact Notation.

The base condition I^b (labelled "Base" in the compact notation) identifies two roles related through a parent relation. The DPO rule specifies that if both roles have permission to the same function, then the permission of the child is deleted. The recursion condition I^r (labelled "Recursion(...)" in the compact notation) goes down the role hierarchy, maintaining the match of the highest role in the hierarchy identified by the base case. As we will see later, the DPO rule will be applied for each step in the hierarchy identified by consecutive matchings of I^r. The following shortcut is used in the compact notation for recursive rules: elements P^b and P^r are hidden, but can be calculated from the numeric labels. In this way, P^b (resp. P^r) is the intersection of the numeric labels in I^b (resp. I^r) and the rule. Morphisms p^b and p^r identify elements with the same numbers. On the other hand, the relation between I^b and I^r (and between two recursive cases) in the compact notation is given by means of the call after "Next:" in the base case. Thus, P^{br} (and

therefore i^{br}) is given by the actual parameters of the recursion call from the base case (that we depict using the identities of nodes and edges, shown before the colon). Note also that morphism p^{br} is given by the assignment of the formal parameters in the recursive condition (i.e. *ra* and *rb*). In a similar way, P^{rr} (and i^{rr}) is given by the actual parameters of the recursion call from the recursive case and p^{rr} by the assignment of the formal parameters in the recursive condition (i.e. *ra* and *rb*). Note how some formal parameters of the recursion may become unused by one of the two recursive calls, and in this case they are just ignored.

4.1 Derivation by DPO Recursive Rule

A DPO recursive rule derivation is made of three steps, each composed by several sub-steps.

First step. The rule is executed for the base case (see Fig. 5). A match $e^b : I^b \to G$ (called *base match*) has to be found for the *base condition* I^b, identifying the starting point in the recursive structure to be processed. Then, a *rule match* $m_1^b : L \to G$ is sought for L, such that $e^b \circ i^b = m_1^b \circ p^b$, and which has to make $(P^b, i^b : P^b \to I^b, p^b : P^b \to L)$ the pullback of $(G, e^b : I^b \to G, m_1^b : L \to G)$, as square (1) in Fig. 5 shows. Then, rule $L \xleftarrow{l} K \xrightarrow{r} R$ is applied once in G, yielding graph H_1. As Proposition 1 shows, match e^b still exists in H_1. If a match $m_2^b : L \to H_1$ is found such that square (2) in Fig. 5 is pullback, we apply again the rule at that match. The operation is repeated until no match from L is found making P^b a pullback object. The output of this step is the base match e^b, together with graph H_n, obtained as a result of the repeated applications of the DPO rule. The execution of the recursive rule finishes if $\nexists e^b$ such that (1) is pullback. The execution continues at step 2 even if the DPO rule is not applied.

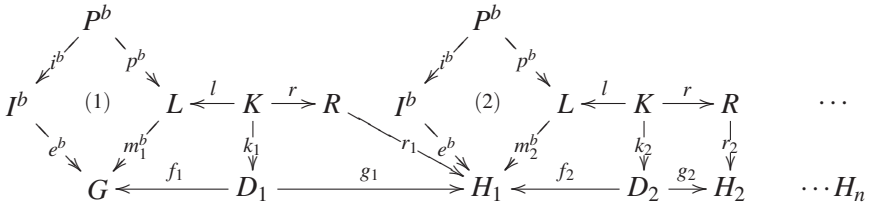


Figure 5: Application of Recursive Rule. Step 1: Base Case.

Second step. The rule is executed for the first step in the recursion. Thus, a match $e_1^r : I^r \to H_n$ (called *recursive match*) is sought such that square (1) in Fig. 6 commutes (i.e. $e^b \circ i^{br} = e_1^r \circ p^{br}$) and makes P^{br} a pullback object. As in the base case, a rule match $m_{1,1}^r : L \to H_n$ is sought for L such that $e_1^r \circ i^r = m_{1,1}^r \circ p^r$ and makes P^r a pullback object, as square (2) shows. If match $m_{1,1}^r$ satisfies the constraints given by Proposition 2 (which guarantee the preservation of e^b), the DPO rule is executed at that match yielding H_{n+1}. As Proposition 1 shows, match e_1^r still exists in H_{n+1}. If an additional match $m_{1,2}^r : L \to H_{n+1}$ is found such that square (3) in Fig. 6 is pullback (and that satisfies Proposition 2), we apply the rule at that match. The operation is repeated until no such rule match is found. In addition, the process is repeated for additional recursive matches $e_1'^r$ commuting with e^b and making P^{br} a pullback object. This can be done as e^b is preserved by the DPO rule applications. The output of this step is a graph H_m resulting from the DPO rule executions, together with the set of recursive matches: $\{e_1^r, e_1'^r, ...\}$. The execution

of the recursive rule stops if $\nexists e_1^r$ such that (1) is pullback. Even if the DPO rule is not applied, the recursive rule continues in step 3.

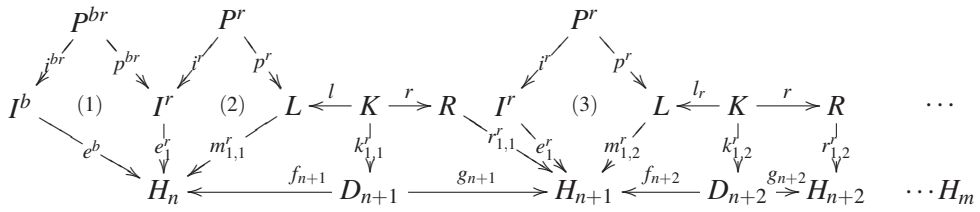


Figure 6: Application of Recursive Rule. Step 2: First Recursive Call.

Third step. We execute the following steps in the recursion. The idea is similar to step 2, but starting from $I^r \xleftarrow{i^{rr}} P^{rr} \xrightarrow{p^{rr}} I^r$ instead of $I^b \xleftarrow{i^{br}} P^{br} \xrightarrow{p^{br}} I^r$. Recursive step $i+1$ is applied for each recursive match provided by previous recursive step, $\{e_i^r, e_i'^r, ...\}$. Each DPO rule application must preserve all these matches, thus each rule match $m_{i+1,j}^r$ must satisfy the conditions of Proposition 2 for each match provided by previous recursive step. The recursive steps are executed as long as a match e_{j+1}^r is found for the next step in the recursion such that square (1) in Fig. 7 is pullback.

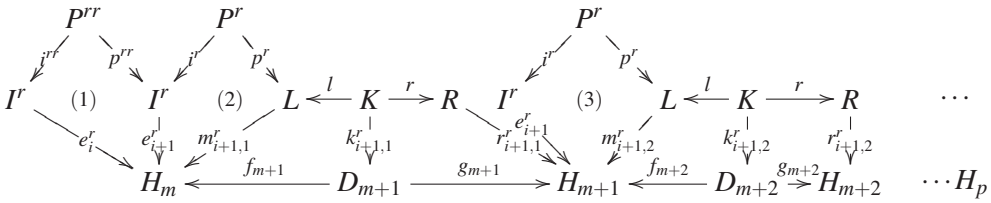


Figure 7: Application of Recursive Rule. Step 3: Successive Recursive Calls.

Definition 5 *(DPO recursive rule derivation) Given a DPO recursive rule* $p_r = (L \xleftarrow{l} K \xrightarrow{r} R, I^b, I^r, (I^j \xleftarrow{i^j} P^j \xrightarrow{p^j} L)_{j\in\{b,r\}}, (I^j \xleftarrow{i^{jr}} P^{jr} \xrightarrow{p^{jr}} I^r)_{j\in\{b,r\}})$*, an object* G*, and a morphism* $e^b : I^b \to G$*. A DPO recursive rule derivation* $G \xRightarrow{p_r, e^b} H_p$ *is built as follows:*

1. *The first step is given by the diagram in Fig. 5, yielding graph* H_n *and base match* e^b*. (written* $G \xRightarrow{(e^b, m_1^b)} H_1 \xRightarrow{*} H_n$ *).*
2. *The second step is given by the diagram in Fig. 6, yielding graph* H_m *and (a possibly empty) set of recursive matches* $\{e_1^r, e_1'^r, ...\}$ *(written* $H_n \xRightarrow{(e_1^r, m_{1,1}^r)} H_{n+1} \xRightarrow{*} \cdots H_k \xRightarrow{(e_1'^r, m_{1,1}'^r)} \cdots H_m$ *)*
3. *The third step is given by the diagram in Fig. 7, for each recursive match coming from the previous application* $\{e_j^r, e_j'^r, ...\}$*, yielding graph* H_p *and a (possibly empty) set of recursive matches* $\{e_{j+1}^r, e_{j+1}'^r, ...\}$ *(written* $H_m \xRightarrow{(e_j^r, m_{j,1}^r)} H_{m+1} \xRightarrow{*} \cdots H_s \xRightarrow{(e_j'^r, m_{j,1}'^r)} \cdots H_p$ *)*

This third step is repeated until the set of newly found recursive matches is empty.

Remarks: The recursive rule execution starts at a unique match e^b. However, in each recursive step, the execution considers all morphisms e'^r_n. In a recursive step, the match is passed between I^b and I^r, and between I^r and I^r. This is because we want to continue the traversal of the structure, even if no rule match for L is found at some step. As the DPO rule is executed as long as possible in every step, and as we seek e^r_{i+1} morphisms as long as possible[1], it is possible to have non-terminating derivations, written $G \overset{p_r, e^b}{\Longrightarrow} \infty$. The DPO recursive rule is applied if the DPO rule is applied at least once.

Definition 3 is modified in a straightforward way, by allowing productions in set P to be either standard DPO rules, or DPO recursive rules. In a derivation, each step can be given by a standard derivation or a recursive one.

Next, we show that morphism e^b can be extended to the resulting graph of applying the DPO rule. The discussion for morphism e^r is similar, so we only present the case for e^b.

Proposition 1 *(Morphism Extension for Recursive Rule) Given the diagram in Fig. 8(a) (with (1) pullback, (3) and (4) pushouts, i^b, p^b, l and $r \in \mathscr{M}$), morphism e^b can be extended to H_1.*

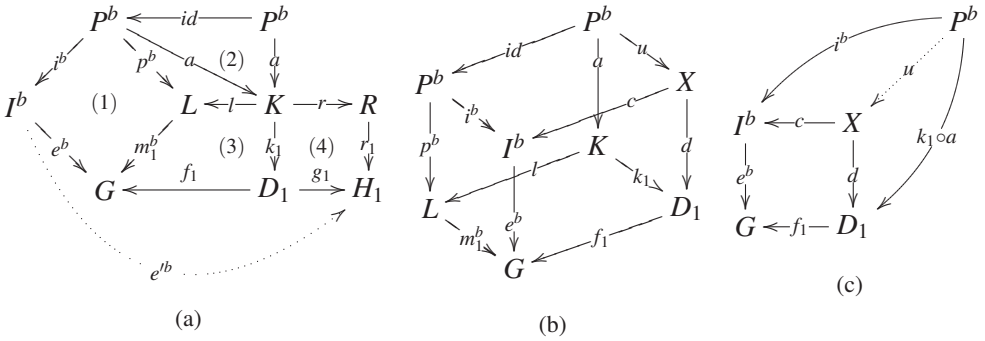


Figure 8: (a) Extension of e^b Morphism. (b) Showing that (3) is a van Kampen Square. (c) Universal Pullback Property.

Proof: First, note that (2) is a pullback. In order to show the existence of a morphism $e'^b : I^b \to H_1$, we use the fact that pushout (3) is a van Kampen square. For that purpose, we build a cube taking (3) at the bottom, and pullbacks (2) and (1) as back-left and front-left faces (see Fig. 8(b)). We close the cube by calculating the pullback of $e^b : I^b \to G$ and $f_1 : D_1 \to G$, given by $(X, c : X \to I^b, d : X \to D_1)$. By the universal property of pullbacks (see Fig. 8(c)), there is a unique arrow $u : P^b \to X$ (as $e^b \circ i^b = f_1 \circ k_1 \circ a$). P^b is the pullback object of k_1 and d by the pullback composition and decomposition lemma. The four lateral faces are pullbacks, and (3) is a pushout along $\mathscr{M}$ ($l, f_1 \in \mathscr{M}$), and therefore a van Kampen square by definition of AHLR

[1] Due to the possible presence of cycles in the graph to be traversed. This can be controlled with NACs in the recursive condition, see the end of the section, and the last example in the paper.

category. Thus the top square is a pushout as well. In particular $c: X \to I^b$ is an isomorphism as the oposite arrow is also an isomorphism. Therefore, we obtain $e'^b = g_1 \circ d \circ c^{-1}$ □

Note that in the first and successive recursive steps, the DPO rule can be applied as long as possible. Moreover, after such rule applications, an additional match $e_i'^r$ is sought in order to repeat the rule applications. Therefore, a DPO rule application in the first recursive step has to preserve e^b (see Figure 6). In a similar way, in recursive step $i+1$ a DPO rule application has also to preserve match e_i^r (see Figure 6) as well as the set of matches output by previous recursive step. The conditions for this preservation are stated in next proposition. We only show the preservation of the base morphism e^b in the first recursion step, as the other cases are similar.

Proposition 2 *(Preservation of Base Match in First Recursive Step) Match $e^b: I^b \to H_n$ is preserved after a standard rule application through match $m^r_{1,1}: L \to H_n$ (as Figure 9(a) shows) if $\exists b^r: P \to P^r$ where $(P, b^b: P \to I^b, b^l: P \to L)$ is the pullback of $e^b: I^b \to H_n$ and $m^r_{1,1}: L \to H_n$ (as Figure 9(b) shows).*

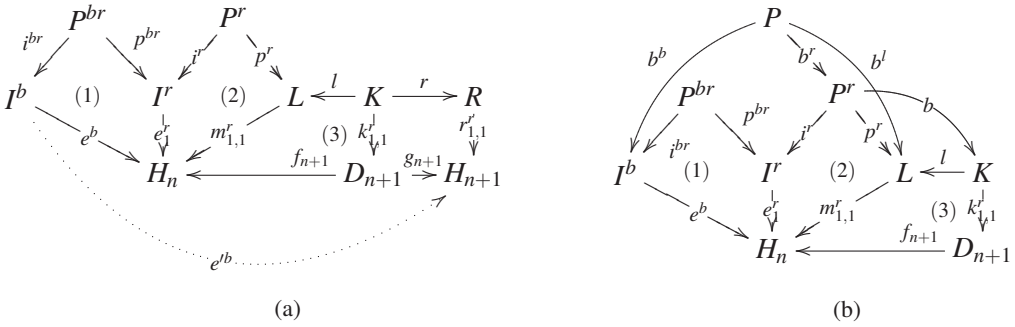


Figure 9: (a) Extension of e^b in Recursive Step One. (b) Condition for Standard Rule Derivation.

Proof: Using the fact that (3) is a van Kampen square, by using the pullback square spawned by P as front left face, taking $b \circ b^r: P \to K$, and then following the construction shown in Proposition 1. □

Remarks: In the general case P is not isomorphic to the pullback object of $p^{br}: P^{br} \to I^r$ and $i^r: P^r \to I^r$. If the conditions stated in this proposition are not satisfied, then the DPO rule cannot be applied.

Example. Fig. 10 shows a derivation of the DPO recursive rule shown in Fig. 4. First, a match of the base condition I^b is identified in G. The match is given by the elements labelled with the same numbers as in I^b, which in addition are coloured. Next, a match of the LHS is sought in G through the elements identified by I^b (i.e. the roles matched by the base condition are the ones used in the math of the LHS). One match is found for which the rule is applied, yielding graph H1. The rule deletes a permission for a role if its parent already defines it (as the base condition identified). Since no other match of the LHS is found in the graph for the base condition, the first recursive step starts. A match of I^r in the graph is found that maintains the match for the role labelled as "1", and identifies the role labelled as "2" with its direct child. Next, a match

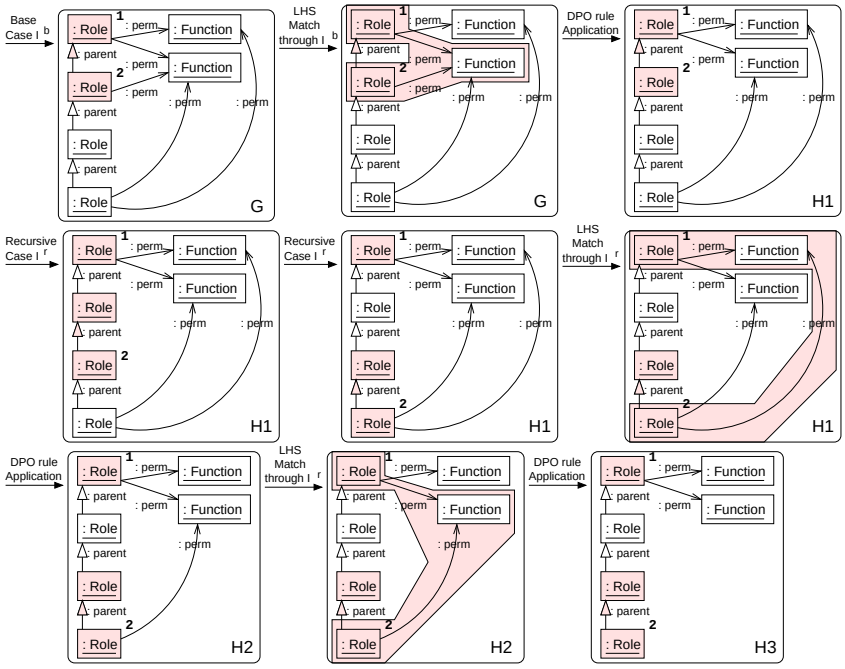


Figure 10: A DPO Recursive Derivation.

of the LHS is sought on the graph through the roles newly labelled "1" and "2", but none is found. Thus, a new step in the recursion starts. A new match of I^r is searched in the graph that maintains the match for role "1" and identifies role "2" with its direct child. Now, two matches of the LHS are found (i.e. the parent role has two permissions which role "2" also defines). In a first application of the DPO rule, one of the permissions is deleted yielding graph H2. A second application of the DPO rule at the other match deletes the second redundant permission yielding graph H3. Since neither matches of the LHS nor matches of the recursive condition are found, the derivation concludes.

Application Conditions. Previous derivation has started in the top-most role in the hierarchy. However, the initial match e^b could also have identified other roles as base condition. In order to identify the top-most role, we need a NAC associated to I^b, which forbids the match if the role has some parent. We extend recursive rules with NACs for I^b, I^r and the DPO rule.

Definition 6 *(DPO recursive rule with NACs) A recursive rule with NACs* $p_r = (L \xleftarrow{l} K \xrightarrow{r} R, NAC_L, I^b, NAC_{I^b}, I^r, NAC_{I^r}, (I^j \xleftarrow{i^j} P^j \xrightarrow{p^j} L)_{j\in\{b,r\}}, (I^j \xleftarrow{i^{jr}} P^{jr} \xrightarrow{p^{jr}} I^r)_{j\in\{b,r\}})$ *is made of a a recursive rule* $(L \xleftarrow{l} K \xrightarrow{r} R, I^b, I^r, (I^j \xleftarrow{i^j} P^j \xrightarrow{p^j} L)_{j\in\{b,r\}}, (I^j \xleftarrow{i^{jr}} P^{jr} \xrightarrow{p^{jr}} I^r)_{j\in\{b,r\}})$ *and three sets* $NAC_J = \{(X^J, x_i^J : J \to X_i^J)\}$ *(for* $J \in \{L, I^b, I^r\}$*) of NACs for* L, I^b *and* I^r.

Previous definition modifies the concept of derivation in the following way. A valid match $e^b : I^b \to G$ has to satisfy $NAC(x_j^{I^b})$, $\forall (X_j^{I^b}, x_j^{I^b}) \in NAC_{I^b}$, and similar for I^r and L.

5 Additional Examples

This section shows further examples in the category of attributed typed graphs [EEPT06] (also an AHLR category). In this category, objects are typed graphs, with edge and node attribution.

Model Transformation. Transforming class diagrams into relational data base models is a common case study when studying the expressivity of model transformation languages [EGL+05]. It requires handling complex structures such as inheritance hierarchies. This transformation implies mapping each persistent class to a table, and all its attributes and associations to columns in this table. However, only top-most classes in the inheritance hierarchy have to be mapped into tables; additional attributes and associations of subclasses result in additional columns of the top-most classes. For the present example, we only consider this excerpt of the problem and restrict to transformation of attributes with a primitive data type.

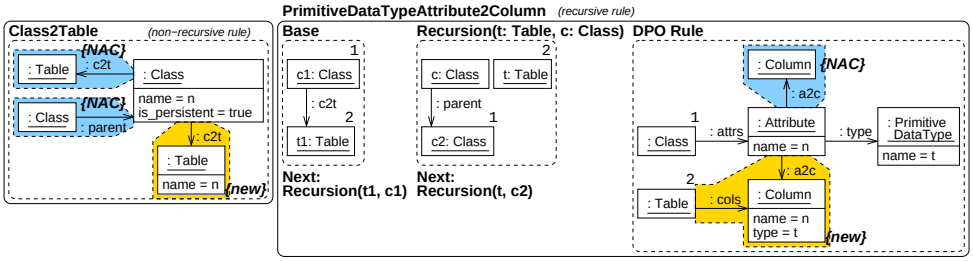


Figure 11: DPO Recursive Rules for Model Transformation.

Using standard DPO rules to perform the transformation would possibly imply the flattening of the hierarchy (see the solutions proposed in [EGL+05]). This is not optimal as it requires modifying the type graph (the meta-model), together with pre- and post-processing rules. We can avoid this flattening by using a DPO recursive rule as the one shown in Fig. 11. The first rule in this figure is a DPO standard rule that creates a table for each top-most class in the hierarchy. The second rule is recursive. The base condition identifies a top-most class in the hierarchy (the one with an associated table). The DPO rule maps a column to each attribute of this class, if such column has not been created before. The column is added to the table mapped to the class. The recursive condition goes down the class hierarchy, maintaining the match of the table. In this way, the application of the rule in the recursive steps creates a column in the table for each attribute of the descendant classes of the base class.

Model Simulation. Fig. 12 shows a recursive rule that updates a network of two-input logical gates when one of the inputs changes. The network of gates can be seen as a recursive structure, which must be traversed to update the output of the gates in each step. The type graph contains a node *Bit*, connected to a *Gate* by relations *in* and *out*. Bits have a self-loop to indicate that their value has been changed externally, and an attribute *value* of type *bit*. Gates have attribute *operation*, an enumerate type with values *or* and *and*, indicating the operation it performs.

In the DPO recursive rule, the base condition looks for a bit that has been changed (i.e. it has a self-loop). The DPO rule updates the output of the gate[2]. An application condition makes the DPO rule applicable only if the value of the output bit has to be updated. We add this attribute

[2] "value=[c, op(a,b)]" denotes that "value" changes from "c" to "op(a,b)" due to the DPO rule application.

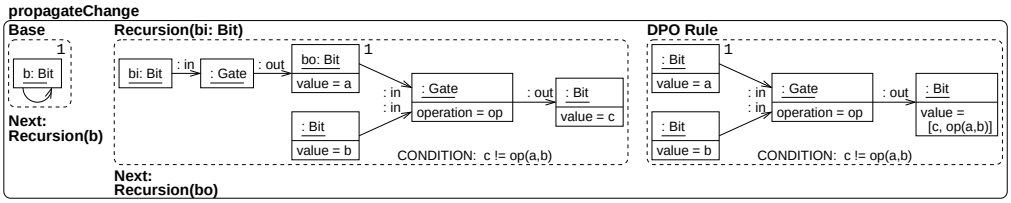


Figure 12: DPO Recursive Rule for Model Simulation.

condition as we may have loops in the network, so the rule must not be applicable if a bit has already the correct value. The recursive condition advances through the network of gates and has a condition which prohibits finding a match if the output gate already has a correct value. Thus, rule application ends when all the bits dependent on the changed bit are updated.

Solving this problem using DPO simple rules would probably imply encoding control elements in the graph to guide rule application. These control elements have to mark the bits to be recalculated, and take care that this calculation occurs in the right order.

6 Conclusions and Future Work

We have presented a novel approach for modelling recursive rules in the DPO approach. The main idea is to provide DPO rules with base and recursive conditions, together with mechanisms to pass the matching between successive recursion steps. The execution mechanism performs a width-first traversal of the recursive structure, while guaranteeing its preservation by the DPO rule applications. We have shown the utility of the approach for structures that can be recursively dealt with, such as inheritance hierarchies, networks of components, etc. We have presented several examples in the areas of model simulation, model optimization and model transformation. The solution of the presented problems using DPO simple rules would imply either the *flattening* of the structure by adding helper edges, or encoding control elements in the host graph to guide rule execution. As we showed, both solutions are not optimal as they imply modifying the type graph and defining pre- and post-processing rules. We believe our DPO recursive rules are a solution to this problem. Moreover, the presented techniques may serve as the basis for a formal rule execution control language with parameter passing.

There are several useful extensions to this approach. The first one is having more than one recursive condition (i.e. more than one I^r). This is useful if the structure to be traversed is not uniform, but it is made of different edge types. The second one is having more than one DPO rule in a recursive rule. This is useful if slightly different actions have to be performed at each step in the recursion, and allows for indirect recursion. It is also worth studying termination and conflicts of DPO recursive rules, as well as different execution policies. Moreover, given a recursive rule, we are investigating the construction of a (possibly infinite) set of standard DPO rules, such that the execution of one of them is equivalent to the execution of the recursive rule.

Acknowledgements: This work has been partially sponsored by the Spanish Ministry of Education and Science with projects MOSAIC (TSI2005-08225-C07-06) and MODUWEB (TIN

2006-09678). The authors gratefully thank the referees for their useful suggestions.

References

[BV06] A. Balogh, D. Varró. Advanced Model Transformation Language Constructs in the VIATRA2 Framework. In *Proc. ACM SAC'06*. Pp. 1280–1287. 2006.

[EEPT06] H. Ehrig, K. Ehrig, U. Prange, G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. Springer, Berlin, Heidelberg, New York, 2006.

[EGL+05] K. Ehrig, E. Guerra, J. de Lara, L. Lengyel, T. Levendovszky, U. Prange, G. Taentzer, D. Varró, S. Varró-Gyapay. Model Transformation by Graph Transformation: A Comparative Study. In *MTiP 2005, (Satellite Event of MoDELS 2005)*. 2005.

[EJ04] N. V. Eetvelde, D. Janssens. Extending Graph Rewriting for Refactoring. In *Proc. ICGT'04*. LNCS 3256, pp. 399–415. Springer, 2004.

[KASS03] G. Karsai, A. Agrawal, F. Shi, J. Sprinkle. On the Use of Graph Transformation in the Formal Specification of Model Interpreters. *JUCS* 9(11):1296–1321, 2003.

[KBC05] A. Kalnins, J. Barzdins, E. Celms. Model Transformation Language MOLA: Extended Patterns. In *Proc. DB&IS'2004*. Volume 118, pp. 169–184. IOS Press, 2005.

[LS04] S. Lack, P. Sobocinski. Adhesive Categories. In Walukiewicz (ed.), *FoSSaCS*. LNCS 2987, pp. 273–288. Springer, 2004.

[LV04] J. de Lara, H. Vangheluwe. Defining visual notations and their manipulation through meta-modelling and graph transformation. *JVLC* 15(3-4):309–330, 2004.

[Min02] M. Minas. Concepts and realization of a diagram editor generator based on hypergraph transformation. *Sci. Comput. Program.* 44(2):157–180, 2002.

[NNZ00] U. Nickel, J. Niere, A. Zündorf. The FUJABA environment. In *Proc. ICSE '00*. Pp. 742–745. ACM Press, 2000.

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

[Roz97] G. Rozenberg (ed.). *Handbook of graph grammars and computing by graph transformation: volume I. foundations*. World Scientific Publishing Co., Inc., 1997.

[SWZ99] A. Schürr, A. J. Winter, A. Zündorf. *The PROGRES approach: language and environment*. World Scientific Publishing Co., Inc., 1999.

[Tae96] G. Taentzer. Parallel and Distributed Graph Transformation. Formal Description and Application to Communication-Based Systems. Shaker Verlag, 1996.

[Wil03] E. D. Willink. A concrete UML-based graphical transformation syntax: The UML to RDBMS example in UMLX. Metamodelling for MDA, York, England, 2003.