Matters of Model Transformation

Eugene Syriani

January 2, 2009

Abstract

Model transformation is at the center of current model-driven engineering research. The current state-of-the-art includes a plethora of techniques and tools for model transformations. One of the most popular approaches is based on graph rewriting. In this report, the most important foundations of graph transformation as well as some of the most efficient techniques for applying these transformations are reviewed. A comparison of the implementation of controlled or programmed graph transformation in various tools is also given. More declarative specifications of graph transformations are discussed. Finally, scalability of model transformation with non graph-based approaches is investigated. The goal of this literature review is to identify open problems and to serve as a starting point for new research.

1 Overview

In model-driven engineering, models and transformations are first-class entities. A model represents an abstraction of a real system, focusing on some of its properties. Models are used to specify, simulate, test, verify, and generate code for applications. In language engineering terms, a model conforms to a *meta-model*. A meta-model defines the abstract syntax and static semantics of a set (possibly infinite) of models. A model is thus typed by its meta-model that specifies its permissible syntax, often in the form of constraints. A common representation of meta-models uses the Unified Modelling Language (UML) Class Diagram notation [1] with Object Constraint Language (OCL) constraints [2]. Model transformation transforms a source model into a target model[1], each conforming to their respective meta-model. Mens and Van Gorp [3] define a transformation to be endogenous if the source and target meta-models are the same and exogenous if they are not. Although model transformation is defined at the meta-model level, it is nevertheless applied on models. Czarnecki and Helsen [4] present a broad classification of features of model transformation. The main features of model transformation involve the following high-level concepts.

Transformation rules are the central elements of a model transformation. A rule is specified on one or more *domain* (for each meta-model). The domain expresses what paradigm the rule uses to operate and access elements of the models. Models (as they appear in the rule) can have graph, term, or string-like structures and can hold variables and constraints. The rules act on either the abstract syntax (using meta-model elements) or a concrete syntax (end-user visualisation). A rule may be applied under certain conditions or from a given context. Rule parametrization is useful for reuse of transformation entities and consequently reduces the size of transformations, in terms of number of rules and complexity.

When executing a transformation, the *location determination* is essential. A deterministic transformation implies that a repeated execution will always lead to the same output. When several choices occur in a non-deterministic transformation, it is important to distinguish concurrent execution from one-point execution. In the former case a rule is applied on all the choices at the same time, while in the latter case it is a applied on only one non-deterministically selected location.

[1] Model-to-code and code-to-model transformations are also model transformations since a program can be considered as a model to some extent.

Rule scheduling is part of inter-rule management. Scheduling can be achieved by explicit control structures (discussed in detail in section 4) or can be implicit due to the nature of the rule specifications. Moreover, several rules may be applicable at the same time. Similar selection mechanisms can be used as in the intra-rule case.

Rules can be *organized* modularly (similar to the concept of package in software programming languages), with an explicit flow of execution, or even as a grammar. Section 4 elaborates on the topic.

Incremental transformations typically come into play when consistency among models is needed. Change-detection and change-propagation mechanisms are then used.

Tracing transformation execution is crucial for model transformation debugging. Traces may be automatically generated in the source or the target or in a separate storage.

A transformation may involve more than two models and be generalized to an *n-to-m relation*. Supporting *multi-directionality* (transformation applied on source(s) or on target(s)) becomes necessary when model transformation is used for model synchronization.

There are over 30 model transformation approaches in the literature. Among these approaches, Czarnecki and Helsen distinguish between:

- *direct-manipulation*, where models offer an API to operate on them,
- *structure-driven*, where transformation is performed in phases or runs,
- *operational*, where meta-models are augmented with imperative constructs,
- *template-based*, often uses meta-code and annotations to direct the transformation,
- *relational*, by declaratively (specifying "what", not "how") mapping between source and target models, often in the form of constraints,
- *graph-transformation-based*, where models are represented as graphs, thus the theory of graph transformation is used to transform models, and
- *hybrid* approaches, combining two or more of the previous categories.

The most widely used model transformation technique with relevant impact in the literature and industry is graph transformation. Graph transformation is the main focus of this review. Section 2 covers the theoretical foundations of this approach, providing an algebraic view on model manipulation. In graph transformation, the most costly operation is to find a subgraph matching the precondition of the transformation rules. Three major algorithms used to perform this task are described in section 3. Section 4 is dedicated to comparing how some of the current tools and approaches transform models using graph transformation. These tools however, only consider unidirectional one-to-one model transformations. Extending graph transformations to model-to-model relations is covered in section 5. Since graph transformation is only one approach to model transformation, we propose two other promising approaches in section 6 and discuss differences. Finally, section 7 concludes this review. In appendix, general questions on model transformation have been addressed.

2 Foundations of Graph Transformation

Graphs are often used to model the state of a system. This allows graph transformation to model state changes of that system. Thus graph transformation systems can be applied in various fields. Graph transformation has its roots in classical approaches to rewriting, such as Chomsky grammars and term rewriting. Operationally, a graph transformation from a graph G to a graph H follows these main steps. First, *choose* a rule composed of a left-hand side (LHS) pattern and a right-hand side (RHS) pattern. Then, *find* an occurrence of the LHS in G satisfying the application conditions of the rule. Finally, *replace* the subgraph matched in G by the RHS. In fact, there are only four possible operations that a graph transformation rule can perform on the host graph[2]: *create* new elements, *read* elements, *update* the attributes of an element, and *delete* an element (the so-called CRUD operations).

[2]Unless specified otherwise, graphs are consider typed, attributed, labelled, and directed.

There are different graph transformation approaches to apply these steps, as described in [5]. Among them is the *algebraic* approach, based on category theory with pushout constructs on the category of graphs. Algebraic graph transformation can be defined using either the *Single-Pushout* (SPO) or the *Double-Pushout* (DPO) approach. Since most of the tools adopt one or the other, we will explain DPO (the "safer" approach) and discuss the differences with SPO. For a full description, the reader is referred to [6].

2.1 Algebraic Graph Transformation

We consider the category **Graphs** [6] to present the major results. In this category, the *objects* are directed graphs[3] in the form $G = (V, E, s, t)$ where V is the set of vertices, E is the set of edges, and $s, t : E \to V$ are the source and target functions respectively. The *morphisms* are graph morphisms in the form of $f : G \to H = (f_V : V_G \to V_H, f_E : E_G \to E_H)$ where the mapping from (nodes and edges of) G_1 to (nodes and edges of) G_2 is total, i.e., $\forall e \in E_G, f_V(s(e)) = s(f_E(e)) \wedge f_V(t(e)) = t(f_E(e))$. The *composition operator* and *identity morphism* of **Graphs** lead to component-wise graph morphisms on nodes and edges.

A *pushout* over morphisms $k : K \to D$ and $r : K \to R$ is defined by a pushout object H and morphisms $n : R \to H$ and $g : D \to H$ such that diagram (2) in Figure 1(a) commutes.

A *graph transformation rule* $p : (L \xleftarrow{l} K \xrightarrow{r} R)$, called *production*, is composed of a pair of injective total graph morphisms $l : L \leftarrow K$ and $r : K \to R$ where L, K, and R are respectively the LHS, interface, and RHS of p. In this case, K represents what subgraph to preserve, being the common part of L and R. We can now formally define a graph transformation.

Let $p : (L \xleftarrow{l} K \xrightarrow{r} R)$ be a graph production, D a context graph (which includes K), and $m : L \to G$ a total graph morphism called *match*. Then a *DPO graph transformation* $G \overset{p,m}{\Rightarrow} H$ from G to H is given by the DPO diagram of Figure 1(a), where (1) and (2) are pushouts in the category **Graphs**. This is also known as *direct derivation*.

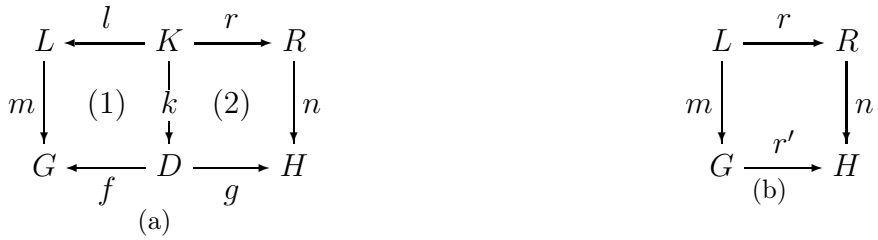

Figure 1: DPO (a) and SPO (b) constructions

In the DPO approach, a transformation rule can thus be applied in the following steps:

1. Find a match $M = m(L)$ in G.
2. Remove $L - K$ (the elements to be deleted) from M such that the *gluing condition* $(G - M) \cup k(K) = D$ still holds.
3. *Glue* $R - K$ (the elements to be created) to D in order to obtain H.

To build the context graph D, the gluing condition comprising the *identification condition* and the *dangling condition* has to be satisfied. In general, L does not need to be isomorphic to M. This is a problem since elements from L cannot be unambiguously identified. The identification condition requires that all elements in $m(R - L)$ (to be deleted) have only one pre-image in L. Another problematic situation is the presence of dangling edges, where the production deletes the source or the target of an edge outside the scope of the LHS. Therefore the dangling condition requires that when p specifies the deletion of a node, it should also delete all its incident edges.

[3] All the results of this section can be generalized to the category of typed, attributed, labelled, and directed graphs.

2.1.1 Concurrency

Assume a graph transformation system with a set of productions $P = \left\{p_i : (L_i \overset{l_i}{\leftarrow} K_i \overset{r_i}{\rightarrow} R_i)\right\}$. Given a host graph G, the rule $p_i \in P$ is applicable if the context graph D_i of pushout (1) in Figure 1(a) exists (i.e., the gluing condition is satisfied). Suppose rules $p_1, p_2 \in P$ are both applicable. Applying p_1 and p_2 in a parallel system allows the two transformations to take place simultaneously. On a sequential system however, their atomic CRUD operations have to be interleaved arbitrarily. Meanwhile, under what conditions can p_1 and p_2 be applied concurrently?

To answer this question, the literature defines the notion of direct derivation independence [5]. Let $d_1 = (G \overset{p_1,m_1}{\Rightarrow} H_1)$ and $d_2 = (G \overset{p_2,m_2}{\Rightarrow} H_2)$ be two direct derivations. d_1 and d_2 are *parallel independent* if they do not conflict: p_2 can still be applied after the application of p_1 and vice-versa. Using DPO this can be formulated as $m_1(L_1) \cap m_2(L_2) \subseteq m_1(l_1(K_1)) \cap m_2(l_2(K_2))$. Therefore, neither of them can delete elements matched by the other. Thus, d_1 and d_2 may only overlap on the elements preserved by both derivations.

On the other hand, two consecutive direct derivations d_1 and d_2 are *sequential independent* if they are not causally dependent: applying first p_1 on G followed by p_2 or p_2 then p_1 leads to the same result. Using DPO this can be formulated as $n_1(R_1) \cap m_2(L_2) \subseteq n_1(r_1(K_1)) \cap m_2(l_2(K_2))$. Therefore, d_2 may not delete elements preserved by d_1 and cannot use any element created by d_1.

The conditions for interleaving d_1 and d_2 is formulated by the *Local Church-Rosser theorem*. Referring to Figure 2(a), the theorem states the following two conditions:

- If $G \overset{p_1,m_1}{\Rightarrow} H_1$ and $G \overset{p_2,m_2}{\Rightarrow} H_2$ are parallel independent, then there exists a graph X and two direct derivations $H_1 \overset{p_2,m_2'}{\Rightarrow} X$ and $H_2 \overset{p_1,m_1'}{\Rightarrow} X$ such that the pair $G \overset{p_1,m_1}{\Rightarrow} H_1 \overset{p_2,m_2'}{\Rightarrow} X$ and the pair $G \overset{p_2,m_2}{\Rightarrow} H_2 \overset{p_1,m_1'}{\Rightarrow} X$ are each sequential independent.
- If two direct derivations $G \overset{p_1,m_1}{\Rightarrow} H_1 \overset{p_2,m_2'}{\Rightarrow} X$ are sequential independent, then there exists a graph H_2 and two direct derivations $G \overset{p_2,m_2}{\Rightarrow} H_2 \overset{p_1,m_1'}{\Rightarrow} X$ such that $G \overset{p_1,m_1}{\Rightarrow} H_1$ and $G \overset{p_2,m_2}{\Rightarrow} H_2$ are parallel independent.

The condition for parallelising d_1 and d_2, is formulated by the *Parallelism theorem*. Referring to Figure 2(b), the theorem states that, given two productions p_1 and p_2, a parallel derivation $G \overset{p_1+p_2,m}{\Rightarrow} X$ exists if and only if there exists sequential independent direct derivations $G \overset{p_1,m_1}{\Rightarrow} H_1 \overset{p_2,m_2'}{\Rightarrow} X$.

Figure 2: Derivations of (a) Local Church-Rosser and (b) Parallelism theorems

These two theorems answer the question on the conditions for applying rules concurrently. Two graph transformations can be applied in an arbitrary order provided they are parallel independent. In this case, they can be applied in parallel via a parallel graph transformation. If two rules are parallel dependent they form a *critical pair*.

2.1.2 Algebraic Graph Transformation using SPO

In SPO, a production $r : L \rightarrow R$ is an injective partial graph morphism r. A partial graph morphism from a graph A to a graph B is a total graph morphism from a subgraph of A to B. L and R denote respectively

the LHS and RHS of p. Given the match $m : L \to G$ as a total graph morphism, a *SPO graph transformation* $G \overset{r,m}{\Rightarrow} H$ from G to H is given by the pushout diagram of Figure 1(b) in the category of graphs with partial morphisms.

The main difference between the SPO and the DPO approach is how they handle the identification and dangling problems. DPO prevents the rule application in both situations, whereas SPO implicitly deletes the problematic nodes. For this reason, DPO rules are invertible and SPO rules are not.

Similar concurrency properties hold for SPO. In fact, a SPO production can be translated to a DPO production defining K as a subgraph of L. However, the reverse translation is not always possible.

2.1.3 Application Conditions

Graph transformation defines the transformation of models at some level of abstraction. The LHS is also called the positive application condition (PAC) since it determines the pattern to be *found* in the host model. Nevertheless, in lots of applications, it is often convenient to specify what pattern should *not* be found. This is referred to as negative application condition (NAC) [7].

A graph transformation rule is then extended with the notion of *application condition*. A production with application condition $\hat{p} = (L \overset{p}{\to} R, A(p))$ consists of a graph transformation rule p and an application condition of $A(p)$. The authors of [7] distinguish the PACs from the NACs in $A(p)$ as sets of total graph morphisms representing positive and negative constraints. In practice, the LHS implicitly contains the PACs, but NACs should be explicitly specified. The production $\hat{p}$ is said *applicable* if, given a match $m : L \to G$, m satisfies all positive and negative constraints of $A(p)$. Assume $A(p)$ only consists of a NAC with negative constraint $\bar{p} : L \to \bar{L}$. $\hat{p}$ is applicable if there is no total graph morphism $\bar{m} : \bar{L} \to G$ such that the composition $\bar{m} \circ \bar{p} = m$ holds as depicted in Figure 3.

Extensions of the DPO approach with NAC have been proposed in [7] and parallel and sequential independence have been adapted accordingly.

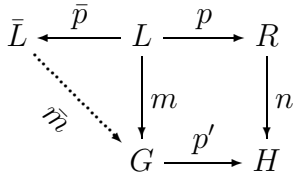

Figure 3: Application of a production with NAC

2.2 Hierarchical Graph Transformation

As mentioned previously, there are other approaches to graph transformation than the algebraic one. For example, in *hyperedge replacement* graph transformations, the LHS and RHS are hypergraphs. The transformation is applied on a hyperedge, which is replaced by an arbitrary hypergraph with designated attachment nodes specified from the LHS.

Drewes et al. have extended the DPO approach for hierarchical graphs using hyperedge replacement transformations [8]. This approach transforms hypergraphs where some hyperedges (called *frames*) can *contain* hypergraphs or variables. They denote transformations on these nested hypergraphs as hierarchical graph transformations. A hierarchical graph $G = \left\langle \hat{G}, F_G, cts_G \right\rangle$ consists of the graph $\hat{G}$ at the root of the hierarchy, the set of frames $F_G \subseteq E_G$ (subset of the hyperedges of G), and a content function cts_G assigning to each frame $f \in F_G$ either a (hierarchical) hypergraph or a variable (symbol). They extend the notion of graph morphism to hierarchical graph morphism from G to H with mappings on root graphs, frames, variables, and contents. The category $\mathbb{H}$ of hierarchical graphs without variables is thus defined. A pushout with injective hierarchical morphisms in $\mathbb{H}$ is defined in the same way as in **Graphs** with injective matches on $\hat{G}$ and on the hierarchical hypergraphs

in F_G recursively. Because the match morphisms are injective, only the dangling condition is necessary to glue (the identification problem is handled like in SPO). In order to handle hierarchical graph transformation with variables, assignment on variables of the LHS are treated as morphism satisfying the dangling condition.

The hierarchical graph transformation defined above does not completely abide to DPO because of the injective morphisms $L \leftarrow K$ and $K \rightarrow R$. As described in [8], hierarchical graph transformation allows encapsulation of local transformation of graphs. This brings the graph transformation paradigm closer to a programming paradigm having the possibility of "storing" patterns inside variables.

In the *node replacement* approach [9], hierarchy is added to graph transformation by letting nodes hold directed graphs.

3 Graph Rewriting Kernel

The core challenge in a graph rewriting system is how the LHS pattern of a transformation rule is found in the host model. In fact, the time efficiency of the transformation process depends highly on the pattern matching algorithm. It is well-known that subgraph isomorphism matching is an NP-complete problem. Nevertheless, since, in graph transformation approaches, graphs are typed, attributed, labelled, and directed, the search space can be drastically reduced to find a subgraph matching the LHS of a rule. Since subgraph isomorphism serves as a basis for the matching phase, we look at two suitable algorithms. Incremental pattern matching is another approach we present.

3.1 Ullmann

Ullmann's algorithm [10] is an efficient solution to the subgraph isomorphism problem. Given two undirected graphs $H = (V_H, E_H)$ and $G = (V_G, E_G)$, the Ullmann algorithm tests whether H is a subgraph of G. We denote by $\mathbf{H}$ and $\mathbf{G}$ their respective adjacency matrices. Let $\mathtt{deg} : V \rightarrow \mathbb{N}$ be a function mapping a vertex to its degree: the number of incident edges it is connected to ($\mathbb{N}$ represents here the set of non-negative integers). The algorithm first constructs the $|V_H| \times |V_G|$ binary matrix $\mathbf{M}^*$ such that:

$$\mathbf{M}^*_{vw} = \begin{cases} 1 & \text{if } \mathtt{deg}(v) \leq \mathtt{deg}(w) \text{ for } v \in V_H \text{ and } w \in V_G, \\ 0 & \text{otherwise.} \end{cases}$$

In this notation, $\mathbf{M}^*_{vw}$ denotes the element of $\mathbf{M}^*$ at the row corresponding to the label of v and the column corresponding to the label of w. $\mathbf{M}^*$ therefore represents the matching of all possible node candidates of V_G that are isomorphic to nodes of V_H. The algorithm tries to find a matrix $\mathbf{M}$ such that $\mathbf{M}(\mathbf{MG})^T = \mathbf{H}$ where every row has exactly one 1 and every column has at most one 1. Therefore $\mathbf{M}$ represents the isomorphic mapping of vertices of H to G. The algorithm thus enumerates all possible such matrices, starting from $\mathbf{M}^*$. At each step, a node in V_H (row of $\mathbf{M}$) is assigned one of the matches (in decreasing order of degrees) by setting to 1 the appropriate column and the rest to 0. This depth-first search is optimized with a *refinement procedure* that takes into account neighbouring nodes: a node V_G may only match if all its neighbours also match. This may set other elements of the matrix to 0, hence reducing the search space. After refining $\mathbf{M}$, if there is a row with no 1, the algorithm backtracks and the next potential match is tried. Otherwise, the algorithm continues on the next row of $\mathbf{M}$. Repeated recursively, the algorithm terminates when either a complete match is found or if all possible matches have been exhausted.

Time efficiency depends highly on how sparse $\mathbf{M}^*$ is initially. Because in graph transformation we consider typed, attributed, labelled, directed graphs as models, the number of $\mathbf{M}$ matrices generated through the search is much smaller than in the general case. This requires comparing incoming and outgoing edges, attribute values, and type compatibility to appropriately fill $\mathbf{M}^*$ with 1s. Some approaches also extend the test of the degree of the node to more sophisticated compatibility tests.

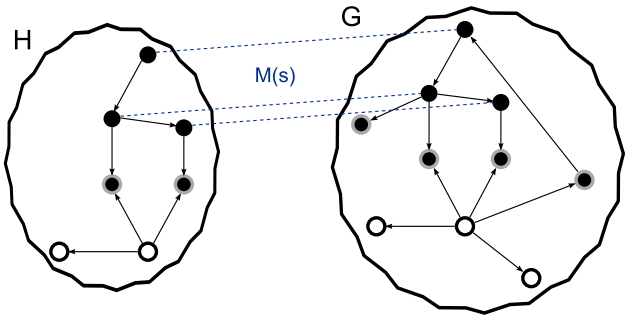

Figure 4: Partial sets for the pruning technique of VF2

3.2 VF2

VF2 [11] is yet another algorithm for the subgraph isomorphism problem. Like in Ullmann's approach, VF2 constructs a search-tree traversing the host graph depth-first and backtracks when the current search-state fails a compatibility test. The algorithm also performs a pruning of the search space during the matching process.

Consider H and G as directed graphs. We denote $M : V_H \rightarrow V_G$ to be the isomorphism node mapping and $M(s)$ holds the set of current matches (v_G, v_H) at search state s (the dashed lines in Figure 4 linking the black nodes). Then let $M_H(s)$ and $M_G(s)$ respectively represent the nodes of V_H and V_G contained in $M(s)$. They are respectively the black nodes in H and in G. VF2 then considers the neighbourhood of $M_H(s)$ by defining $N_H(s)$ and $\bar{V}_H(s)$ (respectively the highlighted and white nodes in Figure 4). $N_H(s) = N_H^{in}(s) \cup N_H^{out}(s)$ where $N_H^{in}(s)$ is the set of nodes adjacent to $M_H(s)$ along incoming edges and $N_H^{out}(s)$ is the set of nodes adjacent to $M_H(s)$ along outgoing edges, not yet in the partial mapping $M_H(s)$. $\bar{V}_H$ is defined as $\bar{V}_H(s) = V_H - M_H(s) - N_H(s)$, representing the nodes not connected to the current mapping. Similar expressions hold for $N_G(s)$ and $\bar{V}_G(s)$.

At each step of the depth-first search, the search-state s is augmented by a candidate pair $p = (v_p, w_p)$ only if it passes a *feasibility test*. (v_p, w_p) is chosen from the ordered list $P(s)$ of candidate pairs. The order suggested by VF2 gives priority to nodes in N_H^{out} and N_G^{out}, then in N_H^{in} and N_G^{in}, and finally (in case of unconnected graphs) in $\bar{V}_H(s)$ and $\bar{V}_G(s)$. The feasibility test run on $s' = s \cup p$ tests three criteria in this order:

1. if the new mapping $M(s')$ is still a valid isomorphism, i.e., edges between v_p and its adjacent nodes in $M_H(s')$ and edges between w_p and its adjacent nodes in $M_G(s')$ correspond,
2. if the number of external edges between $M_H(s')$ and $N_H(s')$ is *smaller than or equal* to the number of external edges between $M_G(s')$ and $N_G(s')$,
3. if the number of external edges between $N_H(s')$ and $\bar{V}_H(s')$ is *smaller than or equal* to the number of external edges between $N_G(s')$ and $\bar{V}_G(s')$,

This way VF2 reduces the search space ensures that no incompatibilities will occur in future search steps. If p fails the feasibility test, the procedure backtracks to the previous state s and tries another candidate. The algorithm terminates when $M(s)$ covers all the nodes of H (success) or when all candidate pairs of $P(s)$ have been tried (failure).

Efficiency-wise, experimental results show that VF2 performs better than Ullmann for larger graphs. Considering $N = |V_H| + |V_G|$ search states to visit in the best case, the time complexity of VF2 is $\Theta(N^2)$. In the worst case, there are $N!$ search states, leading to a time complexity of $\Theta(N!N)$. In both cases, VF2 is a linear order of magnitude more efficient than Ullmann. Furthermore, its spatial complexity is linear, while cubic in the case of Ullmann. These measurements are based on a benchmark [12] considering 10^4 graph experiments and are hence deduced from empirical results.

The major difference between Ullmann and VF2 is that, within one backtracking step, Ullmann compares pairs of adjacent nodes, while VF2 compares a node with its neighbourhood. Moreover, Ullmann's $\mathbf{M}^*$ matrix verifies the semantic compatibility between pairs of nodes in the match, while VF2's feasibility test ensures a correct structure of the match. A combination of VF2 and Ullmann for hierarchical graphs was proposed in [13]. The idea was to merge the two search plans providing containment edges and local edges to denote hierarchy.

The time complexity was thus improved.

3.3 RETE

The initial algorithm [14] was conceived to transform simple chain production systems with integers and strings as primitive entities. An extension of the RETE-algorithm to handle graphs and implement graph grammars was proposed in [15]. Unlike the previous two, this approach does not deal with subgraph isomorphism. Instead, it compiles the transformation rules into a network (nodes and edges) and loads the matches in memory. The graph transformation tool VIATRA is currently the only tool that uses the RETE algorithm for model transformation. They have extended the approach to handle a more expressive and complex pattern language [16]. These patterns resemble the UML class diagram formalism, however they do not handle arbitrary constraints on patterns. The proposed RETE approach in [15] handles typed, attributed, labelled host graphs.

Given a set of productions (or graph transformation rules), the first phase is to compile the LHS of all the productions into a RETE network. The RETE-network is a directed unlabelled graph composed of five different node types:

- **RETE-node** is the root of the RETE network,
- α**-V-node** tests for a single node (type, attribute values),
- α**-E-node** tests for an edge type (type, attribute values),
- β**-node** joins two subgraph instances when they have nodes in common and each of these instances is kept in the β_1-memory or β_2-memory,
- p**-node** holds subgraph instances that satisfy the LHS of rule (or production) p in its p-memory.

The compilation process generates a single RETE-node connected to all α-V-nodes and α-E-node, respectively one for each unique node and edge for all LHS elements of all productions. These α-nodes are either connected directly to a p-node (in case the LHS of p consists of one node) or to a β-node. Then, for every possible unique subgraph (in all the LHSs), a β-node is generated taking a combination of two α/β-nodes as input. Whenever a β-node corresponds to a complete LHS of a production p, it is connected to a corresponding p-node.

Once the RETE-network is built, every node and edge of the host graph is fed into the RETE-node. These data elements are then propagated through the network. When they pass the α tests, they are passed onto the β-nodes which, in turn, propagate the data through the β sub-network. Some of the subgraph instances of the host graph will finally reach the p-nodes. The union of all p-memories is called the conflict set. It represents all the applicable rules and their matches. Depending on the rule organisation and rule scheduling mechanism of the transformation system, the appropriate rule(s) and match(es) is selected (see section 4). To apply the RHS of the selected rule, tokens are generated to create, delete, or update (delete then create) an element. The tokens are propagated to every node in the RETE-network in order to modify its memory state.

In graph transformation, the RETE algorithm is used for incremental transformation. A modification of the model (host graph) is notified to the network through token passing. The LHS is required to be connected, although adding dummy edges can compensate. β-nodes can not merge subgraphs with common edges, so the host graph shall not contain hyperedges. The RETE approach gains a lot of efficiency in time [17] at the price of memory.

4 Controlled Graph Transformation

Since a graph transformation system is composed of graph transformation rules, it is important to specify how these rules are organized. Blostein et al. [18] classify graph transformation organization in four categories. An *unordered* graph-rewriting system simply consists of a set of graph-rewriting rules. Applicable rules are selected non-deterministically until none apply anymore. A *graph grammar* consists of the rules, a start graph, and terminal states. Graph grammars are used for generating language elements: starting from the start graph, apply rules until a termination state is found. They are also used for language recognition: starting from the empty graph, find a sequence of rules that lead to the start graph. In *ordered* graph-rewriting systems, a

control mechanism explicitly orders rule application of a set of rewriting rules. Examples are: priority-based, layered/phased, or with an explicit workflow structure. In this case, the termination condition is specified by the control's final states. Most graph-rewriting systems are partially ordered: applicable rules are chosen non-deterministically while following the control specification[4]. *Event-driven* graph-rewriting systems have recently gained popularity (see section 5.1). In these transformation systems, rule execution is triggered by external events.

Controlled (or programmed) graph rewriting is the key for scaling graph transformation to real-life industrial applications. Controlled graph transformation imposes a control structure over the transformation entities (transformation rules) to have a stricter ordering over the execution of a sequence of rules. This allows for more efficient implementations by providing search plans and pattern caching based on the given order. Initially proposed in [5] and later extended in [19] and then [20], graph transformation control structure primitives may exhibit the following properties:

- *atomicity*: either all rules succeed or they all fail,
- *sequencing*: apply rules one after the other,
- *branching*: execution of a sub-structure based on a condition,
- *looping*: apply rules iteratively,
- *non-determinism*: non-deterministic ordering of rule application,
- *recursion*: ability of a control structure to call itself,
- *parallelism*: apply rules in parallel,
- *back-tracking*: explicit roll-back mechanism, and
- *hierarchy*: in the sense of rule nesting.

In the sequel, we propose to compare the approaches of some of the relevant scalable graph transformation tools that exist today[5]. Table 1 summarizes this comparison.

4.1 ProGReS

The Programmed Graph Rewriting System (ProGReS) was the first fully implemented environment to allow programming through graph transformations [21, 22, 23]. The control mechanism is a textual imperative language. A rule in ProGReS has a boolean behaviour indicating whether it succeeded or not. Among the imperative control structures it provides, rules can be conjuncted using the `&` operator. This allows applying a sequence of rules in order. Branching is supported by the `choose` construct, which applies the first applicable rule following the specified order. ProGReS allows non-deterministic execution of transformation rules. `and` and `or` are the non-deterministic duals of `&` and `choose` respectively by selecting in a random order the rule to be applied. With the `loop` construct, it is possible to loop over sequences of (one or more) rules as long as it succeeds.

A sequence of rules can be encapsulated in a `transaction` following the usual atomicity, isolation, durability, and consistency (ACID) properties. The underlying database system where the models are stored is responsible for ensuring the first three properties. An implicit back-tracking mechanism ensures consistency however. Hence, ProGreS offers two kinds of back-tracking: data back-tracking (with undo operations) and control flow back-tracking [24]. When a rule r' fails in a sequence in the context of a transaction, the control flow will back-track to the previously applied rule r. The data back-tracking mechanism undoes the changes performed by the transformation of r. If r is applicable on another match, it applies the transformation on it and the process continues with the next rule (possibly r'). If r has no further matches, two cases arise. If r was chosen non-deterministically from a set of applicable rules, a non-previously applied rule is selected from this set. Otherwise, the process back-tracks recursively to the rule applied before r. Sequences and transactions can be named allowing recursive calls. The module concept provides a two-level hierarchy in the control flow structure by encapsulating a sequence of transactions.

[4]Note that in graph transformation, non-determinism is implemented using repeatable pseudo-random number generators

[5]This is by no means an exhaustive list.

4.2 AToM^3

AToM^3 is a tool for meta-modelling, multi-formalism modelling, and model transformation [25]. Model transformation can be performed on models conforming to a product of meta-models[6]. Since models are represented as abstract syntax graphs (ASGs), model transformation is performed through graph transformation. It was the first tool to provide a meta-modelling layer in graph transformations.

The control mechanism is limited to a priority-based transformation flow. The transformation system is a graph grammar consisting of graph transformation rules that can be assigned priorities. The rules are applied following the priority ordering: if a rule with higher priority fails, then the rule with the next lower priority is tried. If a rule succeeds, the transformation process starts back at the highest priority rule. These iterations go on until no more rules are applicable. When more than one rule with the same priority is applicable, one of them is chosen randomly, or the user chooses one interactively, or they are applied in parallel. For the latter option, AToM3 does not support overlapping rules conflict detection. It is also possible to divide transformations in layers by sequencing graph grammars – without priorities.

4.3 GReAT

GReAT (for Graph Rewriting And Transformation language) is the model transformation language for the domain-specific modelling tool GME [26]. GReAT's control structure language uses a proprietary asynchronous dataflow diagram notation where a production is represented by a "block" (called *Expression* in [26]). Expressions have input and output interfaces (*inports* and *outports*). They exchange packets: node binding information. The in-place transformation of the host graph thus requires only packets to flow through the transformation execution. Upon receiving a packet, if a match is found, the (new) packet will be sent to the output interface. Inport to outport connections depict sequencing of expressions in that order.

Two types of hierarchical rules are supported. A *Block* forwards all the incoming packets of its inport to the target(s) of that port connection (i.e., the first inner expression(s)of the *Block*). On the other hand, a *ForBlock* sends one packet at a time to its first inner expression(s). When the *ForBlock* has completely processed the packet, the next packet is sent iteratively. Branching is achieved using *Test* expressions. *Test* is a special composite expression holding *Case* expressions internally. A *Case* is given in the form of a rule with only a LHS and a boolean condition on attributes. An incoming packet is tested on each *Case* and every time the *Case* succeeds, it is sent to the corresponding outport. If a *Case* has its *cut* behaviour enabled, the input will not be tried with the subsequent *Cases*. When an outport is connected to more than one inport or if multiple *Cases* succeed in a *Test* (also one-to-many connection), the order of execution of the following expressions is non-deterministic. To achieve recursion, a composite expression (*Block*, *ForBlock*, or *Test/Case*) can have an internal connection to a parent or ancestor expression (in terms of the hierarchy tree).

4.4 VMTS

The controlled graph rewriting system of VMTS is provided by the VMTS Control Flow Language (VCFL) [19], a stereotyped UML Activity Diagram. In this abstract statemachine a transformation rule is encapsulated in an activity, called *step*. Sequencing is achieved by linking steps; self loops are allowed. Branching in VCFL is a *decision step* conditioned by an OCL expression. Chains of *steps* can thus be connected to the *decision*. However at most one of the branches may execute. The *steps* connected to the *decision* should then be non-overlapping (this is checked at compile-time). A branch can also be used to provide conditional loops and thus support iteration.

Steps can be nested in a *high-level step*. A primitive step ends with success when the terminating state is reached and with failure when a match fails. However, in hierarchical steps, when a decision cannot be found at the level of primitive steps, the control flow is sent to the parent state or else the transformation fails. As in GReAT, recursive calls to *high-level steps* is possible. A *fork* connected to a *step* allows for parallelism and a *join*

[6] Cross meta-modelling is commonly referred as multi-formalism modelling.

Property	PROGRES	ATOM^3	GREAT	VMTS
Control Structure	Imperative language	Priority ordering	Data flow	Activity diagram
Atomicity	`transaction`, rule	Rule	Expression	`Step`
Sequencing	`&`	Implicit	Yes	Yes
Branching	`choose...else`	No	`Test / Case`	`Decision step`, OCL
Looping	`loop`	Implicit	Yes	Self loop
Non-determinism	`and`,`or`	Within layer	$1-n$ connection	No
Recursion	Yes	No	Yes	Yes
Parallelism	No	Optional	No	`Fork`, `Join`
Back-tracking	Implicit	No	No	No
Hierarchy	Modularisation	No	`Block`, `ForBlock`	`High-level step`

Table 1: A comparison of the control structure of graph transformation tools

synchronizes the parallel branches. Semantically, parallelism is possible in VMTS but it is not yet implemented [19].

A comparative study of different graph transformation tools can be found in [27]. Another set of tools is compared (though including ATOM^3 and VMTS) based on the solution they provide to a common case-study: the standard benchmark of Class Diagrams to Relational Database Models transformation [28]. The very active field of graph transformation is not restricted to these tools.

5 Graph-based Model-To-Model Relations

The previous sections dealt with graph transformation as "functions": given a host model and a set of transformation rules, produce a *new* model by applying the rules on the host model. In this section, graph transformation is raised to the abstraction level of relations: given two models, specify a relation between them. Because they do not specify any causality, these model-to-model transformations (or relations) are inherently bidirectional. Thus, in one specification, they combine source-to-target and target-to-source transformations. Such declarative graph transformations can then be used for model (co-)evolution, model synchronization, and incremental change propagation between the two models.

5.1 Triple Graph Grammars

Originally, Triple Graph Grammars (TGGs) were inspired by Pair Graph Grammars (PGGs). In 1971, Pratt proposed PGGs [29] to examine string-to-graph translations as a one-to-one context-free mapping. In 1994, Schürr introduced TGGs as graph-to-graph translations and data integration [30]. In contrast with an operational graph grammar, a TGG is not intended to model the editing processes on related graphs (by inserting, deleting, or modifying graph elements), but rather provides a generative description of graph languages and their relationships. A TGG consists of context-sensitive triple productions allowing complex LHS and RHS graphs, as well as a separate correspondence graph for modelling many-to-many relationships. In addition, the correspondence links between the correspondence graph and the LHS and RHS play the role of traceability links that map elements of one graph to elements of the other graph and vice-versa. Furthermore, each correspondence link may carry additional information about the transformation itself.

In a TGG, the graph transformation rules are monotonic: they are non-deleting rules. Following the notation in the commuting diagram of Figure 5(a), a monotonic graph transformation rule $p : L \rightarrow R$ is a graph transformation rule such that $L \subseteq R$ and n consists of all the mappings of m as well as additional mappings restricted from $R - L$ to $H - G$ only. A *monotonic production* is then given by the pushout of Figure 5(a) in **Graphs**.

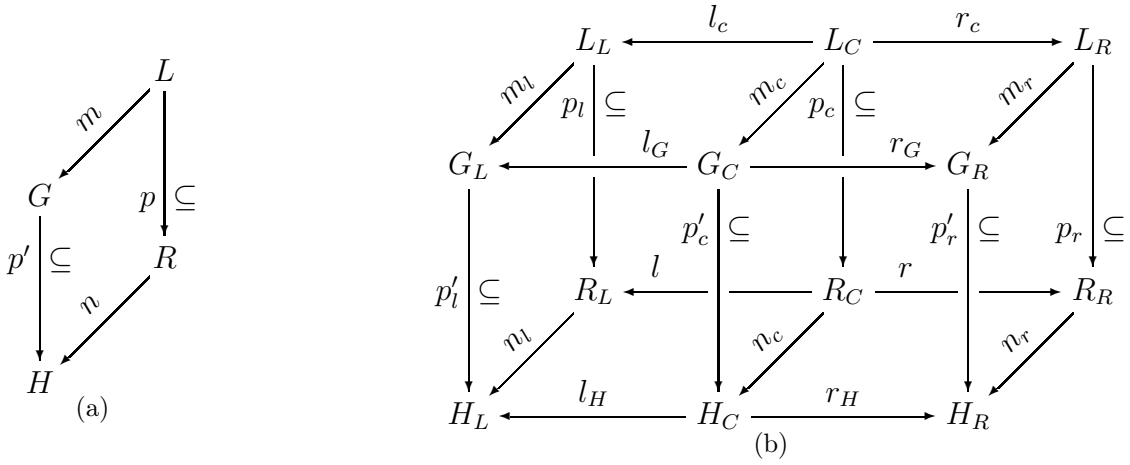

Figure 5: (a) A monotonic production and (b) a TGG production applied on a triple graph. The pushouts should not be confused with the SPO and DPO notation of section 2.

TGGs act on *graph triples* of the form $(G_L \overset{l_G}{\leftarrow} G_C \overset{r_G}{\rightarrow} G_R)$ where L_G, R_G, and C_G are the LHS, RHS, and correspondence graphs respectively. l_G and r_G are graph morphisms allowing m-to-n relationships between L_G and R_G such that every pair of related elements in a subset of $G_L \times G_R$ has a pre-image in G_C.

A *production triple* $p = (p_l \overset{l}{\leftarrow} p_c \overset{r}{\rightarrow} p_r)$ consists of three monotonic productions: $p_l : (L_L \rightarrow R_L)$, $p_r : (L_R \rightarrow R_R)$, and $p_c : (L_C \rightarrow R_C)$ (the left, right, and correspondence pushouts in perspective in Figure 5(b)). Furthermore, $l : (R_C \rightarrow R_L)$ and $r : (R_C \rightarrow R_R)$ are graph morphisms such that the two diagrams at the back of Figure 5(b) are pushouts in **Graphs**. A TGG production is therefore a graph partitioned in three (left, right, and correspondence) graphs. Viewed from another angle, a TGG production contains three graph productions: one operates on a left graph, one on the right graph and one on a correspondence graph. It is this combination of three graph rewriting rules which has to be applied simultaneously that we call "triple graph grammar rule".

Moreover, since a TGG rule is context-sensitive, it is applied on an *axiom* graph triple $G = (G_L \overset{l_G}{\leftarrow} G_C \overset{r_G}{\rightarrow} G_R)$ and produce the result graph $H = (H_L \overset{l_H}{\leftarrow} H_C \overset{r_H}{\rightarrow} H_R)$, as depicted in Figure 5(b).

The presented TGG rules above define a declarative bidirectional transformation from a left graph to a right graph. In a model-driven engineering context, a TGG rule defines a bidirectional relation between two meta-models. The operational semantics of a TGG rule (how the transformation is performed) is described by three kinds of *operational graph transformation rules* (in the sense of section 2) where the LHS and the RHS are triple graphs: creation, deletion, and consistency rules. The former ensures that every new element of one model has a correspondence in the other model. The second makes sure that when an element is removed from one model, its correspondent element(s) is (are) deleted appropriately. The latter enforces the consistency relation between two elements (at the attribute level) by updating the corresponding element. Forward and backwards versions of these rules are generated with an additional traceability rule that creates the correspondence link between unmapped consistent elements of the two models. The (semi-)automatic derivation of some of these "lower-level" transformations is given in [31]. In total seven operational graph transformation rules are generated for each model element, for every TGG rule in the grammar. The operational rules are then given priorities to ensure correct application.

TGGs had a great impact in the graph transformation community allowing declarative and bidirectional graph transformations. For example, in [32] the authors have extended TGGs to handle meta-models with inheritance and parameterized by *events*. Event-driven grammars have been introduced in the context of expressing user interface behaviour. Formalized in the DPO approach, a TGG was used to relate a model's concrete syntax (its visual representation to the user) to its abstract syntax (the graph model). Triple graph grammars have also been extended to multi-graph grammars [33] where an arbitrary number of models can be related. MOFLON [34] is the main model transformation tool that supports TGGs.

5.2 Pattern-based Transformation

As recently pointed out in [35], TGGs do not support NACs. Also, it is not clear how arbitrary attribute manipulation is handled in TGGs. Furthermore, a TGG is context-sensitive and assumes an axiom triple graph as context. This induces causality between rules, thus TGG rules are not *purely* declarative. For these reasons, pattern-based transformation (PBT) was proposed [36]. PBT is highly inspired by TGG, but their intentions differ.

In PBT, a (model-to-model) specification is a conjunction of *triple patterns* acting as constraints over triples graphs (equivalent concept to graph triples in a TGG). A triple graph TrG relates the two models (graphs) by an intermediate correspondence graph. A triple pattern defines a constraint on the graph triple by specifying positive and negative information (similar to PAC and NAC). There are three types of patterns. The simple pattern (S-Pattern) consists of a negative pre-condition $\overleftarrow{N}$, a positive graph Q, and a negative post-condition $\overrightarrow{N}$. An S-Pattern thus states that Q should be found in TrG whenever $\overleftarrow{N}$ is not; and once Q is found, $\overrightarrow{N}$ should not occur in TrG. The composite pattern (C-Pattern) is an S-Pattern with an additional positive pre-condition $\overleftarrow{P}$. The negative pattern (N-Pattern) simply consists of a negative post-condition.

Given a specification, the patterns are compiled into operational TGG rules. The compilation process of the patterns is divided in two phases. First, *deduction rules* are produced. This generates new patterns which take inter-pattern dependencies into account. The N-Patterns are transformed into post-conditions for the S- and C-patterns. The S- and C-patterns are enriched with further pre- and post-conditions according to their dependencies. From there, forward and backwards operational TGG rules are derived.

Some limitations of this approach include that the derivation process does not allow patterns with both positive and negative post-conditions. Although different from TGG, PBT does not handle complex attribute relationship either. No practical application implements PBTs yet. Bidirectional relational transformation is a very active topic of research in the graph transformation community. Other non-graph-based declarative model-to-model transformation approaches exist, such as: QVT-relational [37] and Tefkat [38].

6 Other Model Transformation Approaches

As outlined in section 1, model transformation approaches are not restricted to graph transformation. First we describe how relational database systems can resolve model transformation using similar concepts as graph transformation. Then we describe a hybrid approach (mixing declarative and imperative aspects) provided by one of today's most used model transformation tool.

6.1 Model Transformation in Relational Databases

Graph transformation as described in the previous sections is performed in memory. This approach scales up to some point as long as both models and transformation process fit in memory. However, for very large models (of the order of 10^6 elements) it is preferable to store them in a database. For that reason, Varró et al. propose in [39] a model transformation approach performed in a relational database management system (RDBMS). Once models are stored appropriately in an RDBMS, the transformation specification consists of views and query statements.

Here, we assume that meta-models are initially specified in a subset of UML class diagrams and models in UML Communication diagrams. The transformation, however, requires the models to be represented in a RDBMS in the following way. From the meta-model, one table per class is generated with a column for a unique identifier. Additionally, one column is created per attribute and per many-to-one association. Many-to-many associations are represented as tables on their own with a column for the source and another for the target. Foreign keys ensure the constraint dependencies for association ends and inheritance. Models are stored as rows filling these tables.

The transformation rules follow the SPO graph transformation approach. A rule is divided in two parts: the *matching phase* and the *modification phase*. For the matching phase, the pre-condition LHS $\uplus$ NAC (weaving overlapping elements) of the rule is considered. The LHS is stored as a single view, *LHS-view*, in the RDBMS. An

inner join is added for every object (node) and every association instance (edge) in the LHS. They are filtered according to the edge constraints of the structure of the pattern. Additional filters are used for specifying the exact matching conditions (total injective graph morphism). Finally, the selection projects only the joined columns. Similarly, *NAC-views* are created for each NAC pattern of the rule. LHS $\uplus$ NAC is stored as a separate view. A left outer join of each NAC-view is performed on the LHS-view and the join condition depicts the overlapping elements. To prevent the NAC to be positively matched, the filters of the view force a null value on the columns of the join conditions. Finally, the selected columns are those of the LHS-view.

The modification phase of a transformation rule is encapsulated in a transaction consisting of a sequence of INSERT, DELETE, and UPDATE statements. This phase starts by deleting edges if LHS $-$ RHS $\neq \emptyset$. An UPDATE statement removes the foreign key of the source of a many-to-one association. A DELETE statement removes a many-to-many association as well as any node. Additional DELETE and UPDATE statements are required to ensure the deletion of dangling edges. Then insertions come into place if RHS$-$LHS $\neq \emptyset$. An INSERT statement creates a many-to-many association as well as new node object. An UPDATE statement creates a many-to-one association. In the RDBMS approach, a model element can have an attribute as a one-to-one association between them. This is why there is no UPDATE statement that modifies the value of an attribute.

An advantage of this approach is that a single rule may be applied in parallel on all its matches. This is achieved by applying the modification phase on all the rows returned by the pre-condition view of the rule. Both matching and modification phases can be optimized with the underlying database system used. For example, to perform SPO-like deletion, it may suffice to allow cascading deletes on associations, if they are represented accordingly in the database. Although applying a transformation in a RDBMS is less efficient than in memory, an optimization in time can be gained by properly creating indexes on columns where a matching occurs.

6.2 ATL

The ATLAS Transformation Language (ATL) is a hybrid model transformation language combining declarative and imperative constructs [40, 41]. It is a programming language with its own compiler and virtual machine. An ATL transformation is defined from (possibly several) read-only source meta-models to one write-only target meta-model.

The transformation specification consists of a set of rules and possibly helpers and external modules. The helpers are similar to OCL helpers: they serve as wrappers in the context of source models elements (since the target model is not navigable). *Operation helpers*, taking input parameters, act as functions. *Attribute helpers* decorate the source model by enriching it with a derived subset of its structure.

A declarative rule is called a *matched rule*, since it is transparent from the internal matching and scheduling algorithms of ATL. A matched rule is composed of a source and a target pattern. The source pattern specifies a set of pairs (t, g) where t is a type from the source meta-model and g is an OCL boolean guard. The target pattern is a set of pairs (t', b) where t' is a type from the target meta-model and b is a binding initializing the attributes or references of t'. (t', b) can be replaced by an *action block* where ATL imperative statements are used to build the target model elements. A rule may refer to other rules. *Standard* rules are applied once for every match, *lazy* rules are applied as many times as they are referred to, and *unique lazy* rules are lazy rules but reuse the target elements they created when applied multiple times. Declarative rules support inheritance as means of reuse and polymorphism. A subrule may only match a subset of the match of its parent, but can extend the creation of target elements. A *called rule* is an imperative procedure which can be invoked from a rule (matched or called) and is implemented either using the ATL imperative language constructs or any other language (but the latter has limited support).

Although declarative rules resemble graph transformation rules with a LHS and a RHS, the procedural semantics of an ATL transformation is quite different from the execution of a graph transformation system on a source model. The transformation starts with a first pass through all the guards to evaluate the helpers. The transformation is executed in the second pass. First, a called rule marked as *entry point* is applied if present, which may trigger subsequent rule applications. Then all the matches from all the standard matched rules are computed. Afterwards, for every match, the target elements are created without evaluating the bindings. At the same time, a traceability link between the rule, its matched source elements, and the new target elements

is established internally. Secondly, all initializations (including bindings) are resolved following the *ATL resolve algorithm*. If referenced, lazy rules are applied too. Then action blocks evaluations follow. The algorithm ends by invoking the called rule marked as *end point*, if present. The order of execution of the standard rules is non-deterministic. Nevertheless, determinism and termination of the algorithm is ensured, provided that no lazy or called rules are used.

The Eclipse Modelling Framework (EMF) has adopted ATL as its language and tool support for model transformation. However, ATL lacks of a formal foundation, unlike graph-based transformation.

7 Conclusion

In this review on matters of model transformation, we gave a highly condensed survey of the results of over 20 years of research in this field. After an overview of a desirable set of model transformation features, we focused on the graph transformation approach. Theoretical foundations of the algebraic approach to graph transformation were introduced. Different tools offer rich and expressive languages for graph-based model transformation. Both time and space efficiency are an inherent challenge in graph transformation. Hence, we have described some of the most efficient techniques for solving the matching problem. Other approaches using databases or a mixture of declarative and operational constructs allow handling more complex model transformations and larger models. Bidirectional and incremental transformations are still not mature enough in this field. A grand challenge is to scale model transformation to industrial-size problems without losing the expressiveness of current approaches. This, while allowing the modelling of behaviour and synchronization of models.

Appendix

What is the use of model transformation?

Model transformation has many purposes. Given a formalism, the meta-model defines the abstract syntax (structure) and the static semantics. Model transformation provides *dynamic semantics* (behaviour) to models of the formalism. When the transformation is endogenous (the source and target meta-models are the same), the transformation is typically a *simulation* of the formalism. In this case, a model transformation describes the operational semantics of the language in the formalism. *Refactoring* is another form of endogenous transformation, typically used for optimizing or evolving the design of models.

When a transformation is exogenous (different source and target meta-models), it is typically used to *translate* models from one formalism into another. For example, a domain-specific modelling formalism may be transformed into a lower (abstraction) level formalism such as Petri-Nets or Statecharts. In this case, the meaning of a model is given by a translational semantics into a behaviourally equivalent model.

When two models are related, they can each co-evolve and thus the initial relationship does not hold anymore. Model transformation can be used to *synchronize* models, specifying a bidirectional transformation or relation between models from different meta-models.

Code generation is another form of exogenous model transformation, considering programs as trees and thus as models. It can also be used to allow the integration of a model in a software application and to make the model executable. Other model transformations are useful to serialize models to persistent storage.

Model transformation has applications in several industrial projects. The automobile, military, airspace, and mobile industry (e.g., Porsche, BMW, Nokia, Motorola) are examples of early adopters of model-driven engineering. However, model transformation still suffers from scaling and correctness problems in an industrial context. My contribution is to induce industrial adoption of model transformation.

Why using graph transformation to perform model transformation?

Graphs are often used to model the state of a system. This allows graph transformation to model state changes of that system. Thus graph transformation systems can be applied in various fields of sciences and engineering. In software engineering, UML class diagrams are often used to express the description of meta-models. This formalism can be abstracted to a graph with nodes (classes) and edges (associations). It is thus natural to consider models as graphs and hence use the theory of graph transformation to perform transformation on them.

Graph transformation has a strong theoretical foundation: the algebraic graph transformation. It is based on category theory and many results from this theory are applied directly in modern model transformation tools. For example, the single and double pushout approaches from the categorical algebraic graph transformation is often implemented as is in tools. Critical pair analysis permit the detection of parallel independent rules. The Local Church-Rosser and Parallelism theorems define the conditions to apply rules concurrently.

Another advantage of using graph transformation to transform models is that it is a rule-based technique. This allows specifying the transformation as a set of operational rewriting rules instead of using imperative programming languages. Model transformation can thus be specified at a higher level of abstraction (hiding the implementation of the matching algorithms), closer to the domain of the models it is applied on.

What are applications of negative application conditions?

A graph transformation rule consists of a left-hand side (LHS), a right-hand side (RHS) and optionally a negative application condition (NAC). The LHS represents a pre-condition pattern to be found in the host graph along with conditions on attributes. The RHS represents the post-condition pattern after the rule has been applied on the matched subgraph by the LHS. The NAC represents what pattern condition shall not be found in the host graph, inhibiting the application of the rule. NACs therefore allow increasing the expressiveness of transformation rules. This makes the individual rules and the transformation process more understandable. Also, allowing negative expressions reduces the number of rules for a given transformation (often by a factor of

two since, on top of the rules necessary to the transformation, additional rules must be specified to prevent the application of some of them). NACs may become very handy to prevent a sequence of derivations to process the graph. For example, when the transformation traverses the graph (or parts of it), making use of NACs can prevent infinite loops.

Why is the notion of hierarchy useful for transformations?

A hierarchical graph is a graph where edges or nodes can contain other graphs nested in them. External edges from an inner graph to an ancestor (in the hierarchy tree) are not allowed. Hierarchical graph transformations are useful when the host model is very large: they allow *locality*. A single rule may be focused on parts of the graph. Rules thus gain in expressiveness in the sense that they allow transformations at different levels of the graph hierarchy (abstraction levels). For example, a modeller having designed a sub-model of the whole model can then specify a transformation only for the part he is interested in.

From an implementation point of view, when hierarchical graphs are transformed by hierarchical graph transformation rules, the rule matching may be performed more efficiently. That is, the search space of the matching process can be drastically reduced, given that the rule is applied in a given context (the parent node).

What is rule scheduling?

Given a graph transformation system with a set of rules, rule scheduling describes in what order the rules will be applied (inter-rule management). In declarative graph transformations, the rule scheduling is implicit and transparent to the modeller which defines a relation between meta-models. On the other hand, operational graph transformations require an explicit scheduling of the transformation rules.

An *unordered* graph-rewriting system simply consists of a set of graph-rewriting rules. Applicable rules are selected non-deterministically until none apply anymore.

A *graph grammar* consists of the rules, a start graph, and terminal states. Graph grammars are used for generating language elements: starting from the start graph, apply rules until a termination state is found. They are also used for language recognition: starting from the empty graph, find a sequence of rules that lead to the start graph.

In *ordered* graph-rewriting systems, a control mechanism explicitly orders rule application of a set of rewriting rules. Examples are: priority-based, layered/phased, or transformations with an explicit workflow structure. In this case, the termination condition is specified by the control's final states. Most graph-rewriting systems are *partially ordered*: applicable rules are chosen non-deterministically while following the control specification.

Event-driven graph-rewriting systems have recently gained popularity. In these transformation systems, rule execution is triggered by external events.

Controlled (or programmed) graph rewriting is the key for scaling graph transformation to real-life industrial applications. Controlled graph transformation imposes a control structure over the transformation entities (transformation rules) to have a stricter ordering over the execution of a sequence of rules. This allows for more efficient implementations by providing search plans and pattern caching based on the given order. Graph transformation control structure primitives may exhibit the following properties: atomicity, sequencing, branching, looping, non-determinism, recursion, parallelism, back-tracking, hierarchy, and possibly a dimension of time.

What is traceability?

Tracing transformation execution is crucial for model transformation *debugging*. It can take the form of snapshots of the models at different steps in the transformation process. Tracing may also be expressed by having backward links from the new model to the initial (or an intermediate) model along the transformation. Traces may be automatically generated in the source or the target or in a separate storage. This is typically useful when the transformation transforms a model in one formalism into a model in another formalism. Traceable transformations are also very important when code synthesis and reverse engineering is needed. For domain-specific

languages, animating a model through a model transformation requires traces between the abstract model and its visual concrete syntax model.

In bidirectional transformation such as triple graph grammars, a rule is specified on a triple graph: the left graph, right graph, and the correspondence graph. The correspondence graph links the other two by specifying the relationship that holds between mapped elements, e.g., a constraint on the attributes of a node. This traceability link between two model elements acts as memory storage of rule applications.

References

[1] Object Management Group (2007) *Unified Modeling Language Superstructure*.

[2] Object Management Group (2006) *Object Constraint Language*.

[3] Mens, T. and Van Gorp, P. (2006) A taxonomy of model transformation. *GraMoT'05*, Tallinn (Estonia), March, ENTCS, **152**, pp. 125–142.

[4] Czarnecki, K. and Helsen, S. (2006) Feature-based survey of model transformation approaches. *IBM Systems Journal, special issue on Model-Driven Software Development*, **45**, 621–645.

[5] Ehrig, H., Engels, G., Kreowski, H.-J., and Rozenberg, G. (1997) *Handbook of graph grammars and computing by graph transformation, Volume 1: Foundations*. World Scientific Publishing Co., Inc.

[6] Ehrig, H., Ehrig, K., Prange, U., and Taentzer, G. (2006) *Fundamentals of Algebraic Graph Transformation* EATCS. Springer-Verlag.

[7] Habel, A., Heckel, R., and Taentzer, G. (1996) Graph grammars with negative application conditions. *Fundamenta Informaticae*, **26**, 287 – 313.

[8] Drewes, F., Hoffmann, B., and Plump, D. (2002) Hierarchical graph transformation. *JCSS*, **64**, 249–283.

[9] Verlinden, N. and Janssens, D. (2004) A framework for NLC and ESM: Local action systems. *6th International Workshop on Theory and Application of Graph Transformations*, February, LNCS, **1764**, pp. 194–215. Springer-Verlag.

[10] Ullmann, J. R. (1976) An algorithm for subgraph isomorphism. *Journal of the ACM*, **23**, 31–42.

[11] Cordella, L., Foggia, P., Sansone, C., and Vento, M. (2004) A (sub)graph isomorphism algorithm for matching large graphs. *TPAMI*, **26**, 1367–1372.

[12] Foggia, P., Sansone, C., and Vento, M. (2001) A database of graphs for isomorphism and sub graph isomorphism benchmarking. In Jolion, J.-M., Kropatsch, W., and Mario, V. (eds.), *Workshop on Graph-Based Representation in Pattern Recognition*, Ischia (Italy), May IAPR-TC15, pp. 176–188.

[13] Provost, M. (2005) Himesis: A hierarchical subgraph matching kernel for model driven development. Master's thesis. McGill University Montréal (Canada).

[14] Forgy, C. L. (1982) Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, **19**, 17–37.

[15] Bunke, H., Glauser, T., and Tran, T.-H. (1991) An efficient implementation of graph grammars based on the RETE matching algorithm. *Graph Grammars and Their Application to Computer Science*, Bremen (Germany), March, LNCS, **532**. Springer.

[16] Bergmann, G., Ökrös, A., Ráth, I., Varró, D., and Varró, G. (2008) Incremental pattern matching in the VIATRA model transformation system. *GRaMot'08*.

[17] Bergmann, G., Horváth, Á., Ráth, I., and Varró, D. (2008) A benchmark evaluation of incremental pattern in graph transformation. *ICGT'08*.

[18] Blostein, D., Fahmy, H., and Grbavec, A. (1996) Issues in the practical use of graph rewriting. In Cuny, J. E., Ehrig, H., Engels, G., and Rozenberg, G. (eds.), *Selected papers from the 5th International Workshop on Graph Grammars and Their Application to Computer Science*, Williamsburg (USA), November, LNCS, **1073**, pp. 38–55. Springer-Verlag.

[19] Lengyel, L., Levendovszky, T., Mezei, G., and Charaf, H. (2006) Model transformation with a visual control flow language. *IJCS*, **1**, 45–53.

[20] Syriani, E. and Vangheluwe, H. (2007) Programmed graph rewriting with DEVS. In Nagl, M. and Schürr, A. (eds.), *AGTIVE 2007*, Kassel (Germany), LNCS, **5088**, pp. 136–152. Springer-Verlag.

[21] Blostein, D. and Schürr, A. (1999) Computing with graphs and graph rewriting. *SPE*, **9**, 1–21.

[22] Zündorf, A. (1994) Graph pattern matching in PROGRES. In Ehrig, H., Engels, G., and Rozenberg, G. (eds.), *Graph Grammars and Their Application to Computer Science*, Williamsburg,USA, November, LNCS, **1073**, pp. 454–468. Springer-Verlag.

[23] Schürr, A., Winter, A. J., and Zündorf, A. (1995) Graph grammar engineering with PROGRES. In Schäfer, W. and Botella, P. (eds.), *Proceedings of the 5th European Software Engineering Conference*, Sitges,Spain, September, LNCS, **989**, pp. 219–234. Springer-Verlag.

[24] Zündorf, A. (1992) Implementation of the imperative / rule based language PROGRES. Aachener Informatik -Berichte 92–38. Department of computer science III, Aachen University of Technology, Germany.

[25] de Lara, J. and Vangheluwe, H. (2002) AToM^3: A tool for multi-formalism and meta-modelling. In Kutsche, R.-D. and Weber, H. (eds.), *FASE'02*, Grenoble (France), April, LNCS, **2306**, pp. 174–188. Springer-Verlag.

[26] Agrawal, A., Karsai, G., Kalmar, Z., Neema, S., Shi, F., and Vizhanyo, A. (2006) The design of a language for model transformations. *SoSym*, **5**, 261–288.

[27] Taentzer, G., Ehrig, K., Guerra, E., de Lara, J., Lengyel, L., Levendovszky, T., Prange, U., Varró, D., and Varró-Gyapay, S. (2005) Model transformation by graph transformation: A comparative study. *MTiP 2005, International Workshop on Model Transformations in Practice*, Montego Bay (Jamaica), October.

[28] Bézivin, J., Rumpe, B., and Tratt, L. (2005). Model transformation in practice workshop announcement. `http://sosym.dcs.kcl.ac.uk/events/mtip05/long\_cfp.pdf`.

[29] Pratt, T. W. (1971) Pair grammars, graph languages and string-to-graph translations. *JCSS*, **5**, 560–595.

[30] Schürr, A. (1994) Specification of graph translators with triple graph grammars. In Tinhofer, G. (ed.), *Graph-Theoretic Concepts in Computer Science*, Heidelberg (Germany), June, LNCS, **903**, pp. 151–163. Springer-Verlag.

[31] Königs, A. and Schürr, A. (2006) Tool integration with triple graph grammars - a survey. In Heckel, R. (ed.), *Proceedings of the SegraVis School on Foundations of Visual Modelling Techniques*, Amsterdam (Netherlands), ENTCS, **148**, pp. 113–150. Elsevier Science Publishers.

[32] Guerra, E. and de Lara, J. (2007) Event-driven grammars: Relating abstract and concrete levels of visual languages. *SoSym*, **6**, 317–347.

[33] Königs, A. and Schürr, A. (2006) MDI: A rule-based multi-document and tool integration approach. *SoSym*, **5**, 349–368.

[34] Amelunxen, C., Königs, A., Rötschke, T., and Schürr, A. (2006) MOFLON: A standard-compliant metamodeling framework with graph transformations. In Rensink, A. and Warmer, J. (eds.), *Model Driven Architecture - Foundations and Applications: Second European Conference*, LNCS, **4066**, pp. 361–375. Springer-Verlag.

[35] Schürr, A. and Klar, F. (2008) 15 years of triple graph grammars. In Ehrig, H., Heckel, R., Rozenberg, G., and Taentzer, G. (eds.), *ICGT'08*, Leicester (UK), September, LNCS, **5214**, pp. 411–425.

[36] de Lara, J. and Guerra, E. (2008) Pattern-based model-to-model transformation. In Ehrig, H., Heckel, R., Rozenberg, G., and Taentzer, G. (eds.), *ICGT'08*, Leicester (UK), September, LNCS, **5214**, pp. 427–441.

[37] Object Management Group (2005) *Meta Object Facility 2.0 Query/View/Transformation Specification.*

[38] Lawley, M. and Steel, J. (2006) Practical declarative model transformation with tefkat. In Bruel, J.-M. (ed.), *Satellite Events at the MoDELS'05 Conference*, Montego Bay (Jamaica), October, LNCS, **3844**, pp. 139–150. Springer-Verlag.

[39] Varró, G., Friedl, K., and Varró, D. (2006) Implementing a graph transformation engine in relational databases. *SoSym*, **5**, 313–341.

[40] Jouault, F. and Kurtev, I. (2006) Transforming models with ATL. *MTiP'05*, January, LNCS, **3844**, pp. 128–138. Springer-Verlag.

[41] Jouault, F. (2006) Contribution è l'étude des langages de transformation de modèles. Ph.d. thesis Université de Nantes.