The Joys of Graph Transformation

Arend Rensink
Department of Computer Science, University of Twente
P.O.Box 217, 7500 AE, The Netherlands
rensink@cs.utwente.nl

February 9, 2005

Abstract

We believe that the technique of graph transformation offers a very natural way to specify semantics for languages that have dynamic allocation and linking structure; for instance, object-oriented programming languages, but also languages for mobility. In this note we expose, on a rather informal level, the reasons for this belief. Our hope in doing this is to raise interest in this technique and so generate more interest in the fascinating possibilities and open questions of this area.

1 Graph Transformation Is Easy

Transformation means changing (literally: shaping) one thing into another. In the case of graph transformation, obviously, the things being changed are graphs. A fundamental assumption in studying such changes is that they are not arbitrary but controlled by some guiding principles, and that these principles can be captured in *rules*. A graph transformation rule (often called a production rule) describes a *kind* of change that will transform certain graphs — those to which the rule is applicable — into others, in a specific way encoded in the rule. A set of production rules is usually called a production system; graphs that are subjected to transformation are often called host graphs.

For us, the interest in this arises from the fact that graphs can be used to model just about any discrete structure — with lesser or greater ease — and that many kinds of dynamic changes in such structures lend themselves quite naturally to a description by graph production rules. This is in particular true of the semantics of object-oriented systems: it is our firm belief that graph transformations are a very natural technique to specify the semantics of such systems (see also Sect. 4). (It should be mentioned that this is not the only, and indeed historically not the original, reason to be interested in graph transformation: another motivation is to characterise graph structures as the end product of a sequence of transformations guided by a given set of rules. As a very simple example, the class of all connected graphs can be characterised in this way. In that context it is not so much the process of change as its result that is of primary interest.)

Although there are many ways to define graph production rules, all rules have certain basic things in common. They always specify changes in a (relatively) small sub-structure, and the change always consists of modifications to that sub-structure, such as taking away parts from it or adding parts to it. For the rule to apply, the first requirement is that the host graph actually contains a sub-structure of the right kind; in fact, if it contains more than one such sub-structure, the rule is applicable in different ways.

This is, of course, a very general description; in practice, there have turned out to be many different useful ways to specify sub-structures and changes. In the past this has led to strong opinions about the relative merits of the various techniques. Fortunately, the purpose of this paper is not to categorise these approaches (for that, the handbook [21] is a

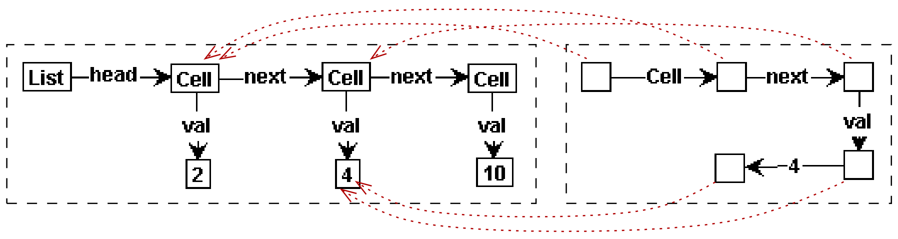

Figure 1: Two graphs with a matching

good source); rather, we want to illustrate the ease with which graph transformation can be used and understood. For this purpose we will concentrate on one, very simple, formalism, which we do not claim to be superior in any formal way but which we do believe to be easily understandable. The approach has been implemented in a tool, called GROOVE, which can be downloaded at `http://groove.sf.net` (see also [17]); all examples in this paper have been produced using the tool.

We will now give some formal definitions. First of all we define the concept of a graph. For this purpose we assume a known set of labels Labels, which contains names with which we will label the edges of our graphs.

Definition 1 (graph) *A graph over* Labels *is a tuple* $\langle \mathsf{Nodes}, \mathsf{Edges}, \mathsf{src}, \mathsf{tgt}, \mathsf{lab} \rangle$, *where* Nodes *and* $\mathsf{Edges} \subseteq \mathsf{Nodes} \times \mathsf{Labels} \times \mathsf{Edges}$ *are finite sets.*

Thus, an edge $\mathsf{e} \in \mathsf{Edges}$ is a triple $(\mathsf{v}, \mathsf{a}, \mathsf{w})$ consisting of a source node $\mathsf{src}(\mathsf{e}) = \mathsf{v}$, a label $\mathsf{lab}(\mathsf{e}) = \mathsf{a}$, and a target node $\mathsf{tgt}(\mathsf{e}) = \mathsf{w}$. If $\mathsf{src}(\mathsf{e}) = \mathsf{tgt}(\mathsf{e})$ we call e a *self-edge*. If G is a graph then we use $\mathsf{G.Nodes}$ and $\mathsf{G.Edges}$ to indicate its components. Note that the definition rules does not allow different edges with the same source nodes, labels and target nodes. If $\mathsf{G.Nodes} \subseteq \mathsf{H.Nodes}$ and $\mathsf{G.Edges} \subseteq \mathsf{H.Edges}$, we call G a *subgraph* of H; we indicate this by $\mathsf{G} \subseteq \mathsf{H}$. We now define what it means for one graph to match another.

Definition 2 (graph matching) *Let* G, H *be graphs. A matching of* G *in* H *is given by a function* $\mathsf{nodeMap} \colon \mathsf{G.Nodes} \to \mathsf{H.Nodes}$, *such that*

$$(\mathsf{v}, \mathsf{a}, \mathsf{w}) \in \mathsf{G.Edges} \textit{ implies } (\mathsf{nodeMap}(\mathsf{v}), \mathsf{a}, \mathsf{nodeMap}(\mathsf{w})) \in \mathsf{H.Edges} \ .$$

We write $\mathsf{nodeMap} \colon \mathsf{G} \to \mathsf{H}$ to denote that $\mathsf{nodeMap}$ is a matching of G in H. We also extend $\mathsf{nodeMap}$ pointwise to edges; i.e.,

$$\mathsf{nodeMap}(\mathsf{e}) = (\mathsf{nodeMap}(\mathsf{src}(\mathsf{e})), \mathsf{lab}(\mathsf{e}), \mathsf{nodeMap}(\mathsf{tgt}(\mathsf{e}))) \ .$$

Example 3 *Fig. 1 shows two graphs, with the notational convention that labels shown on nodes are shorthand for self-edges with those labels. The left hand graph shows a linked-list structure, consisting of* Cell*-labelled nodes (or, more accurately, nodes with* Cell*-labelled self-edges) linked by* next*-labelled edges. The right hand side graph shows another graph over the same label set. Furthermore, the dotted arrows indicate the node map of a matching of the right hand side graph in the left hand side graph. The edge map is not depicted but can be constructed uniquely from the node map (again taking into account that the node labels* Cell *and* $\mathsf{4}$ *on the left hand side actually stand for self-edges).*

Definition 4 (production rule) *A production rule is a tuple* $\langle \mathsf{Lhs}, \mathsf{Rhs} \rangle$ *of graphs. The rule is* applicable *to a graph* G *if there is a matching of* Lhs *into* G.

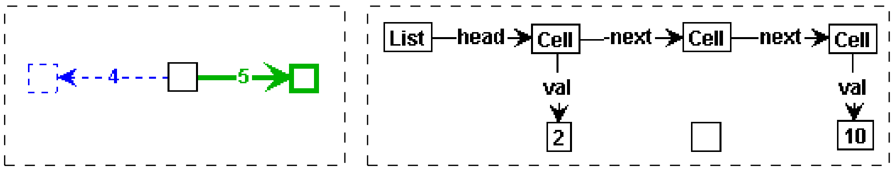

Figure 2: A production rule, and the result of its application to Fig. 1

To define the application of a rule $\langle \mathsf{Lhs}, \mathsf{Rhs} \rangle$ precisely, we identify the sets of elements scheduled to be deleted and those scheduled to be created, as follows:

$$\begin{aligned}
\mathsf{Del.Nodes} &= \mathsf{Lhs.Nodes} \setminus \mathsf{Rhs.Nodes} \\
\mathsf{Del.Edges} &= \mathsf{Lhs.Edges} \setminus \mathsf{Rhs.Edges} \\
\mathsf{New.Nodes} &= \mathsf{Rhs.Nodes} \setminus \mathsf{Lhs.Nodes} \\
\mathsf{New.Edges} &= \mathsf{Rhs.Edges} \setminus \mathsf{Lhs.Edges} \ .
\end{aligned}$$

When we apply a rule to G, the elements that are in Lhs but not in Rhs are deleted from G (or rather, their images under the matching) and elements that are in Rhs but not in Lhs are added (or rather, fresh images are created for them). However, things are complicated by two effects:

- The matching may be non-injective; in particular, there may be $\mathsf{v}_1 \in \mathsf{Del.Nodes}$ and $\mathsf{v}_2 \in \mathsf{Lhs.Nodes} \setminus \mathsf{Del.Nodes}$ such that $\mathsf{nodeMap}(\mathsf{v}_1) = \mathsf{nodeMap}(\mathsf{v}_2)$;
- The matching may be non-surjective on the incident edges of a node scheduled to be deleted; in particular, there may be $v \in \mathsf{Del.Nodes}$ and $\mathsf{e} \in \mathsf{G.Edges} \setminus \mathsf{nodeMap}(\mathsf{Del.Edges})$ such that $\mathsf{G.src}(\mathsf{e}) = \mathsf{nodeMap}(\mathsf{v})$ or $\mathsf{G.tgt}(\mathsf{e}) = \mathsf{nodeMap}(\mathsf{v})$. (Such an edge e is often called a *dangling edge*.)

These problems are resolved by specifying that deletion always wins out: that is, $\mathsf{nodeMap}(\mathsf{v}_1)$ and e in the scenarios above will both be deleted from G. This can have unexpected effects, as the example below will show.

Example 5 *Fig. 2 shows a graph production rule, where the left hand side and right hand side are combined into one graph, with the following notational conventions:*

- $\mathsf{Del.Nodes}$ *and* $\mathsf{Del.Edges}$ *are depicted as thin, dashed (blue) nodes and edges;*
- $\mathsf{New.Nodes}$ *and* $\mathsf{New.Edges}$ *are depicted as wide (green) nodes and edges;*
- *All other nodes and edges are in the intersection of* Lhs *and* Rhs.

In other words, Lhs *consists of the thin (continuous and dashed) nodes and edges and* Rhs *consists of the continuous (thin and wide) nodes and edges.*

Intuitively, the rule in Fig. 2 specifies that a $\mathsf{4}$*-labelled edge is to be deleted, together with its target node, and a* $\mathsf{5}$*-labelled edge is to be created to a new node. However, the only matching of* Lhs *in the left hand graph of Fig. 1 maps both rule nodes to the same graph node; moreover, this graph node also has an incoming* val*-edge, about which the rule says nothing. The result is that the node is deleted, with its incoming* val*-edge; even more curiously, the* $\mathsf{5}$*-edge that was just created is deleted as well, but the node that was also just created is preserved. The result is also shown in Fig. 2.*

The following algorithm defines how to apply a graph production rule $\langle \mathsf{Lhs}, \mathsf{Rhs} \rangle$ to a host graph G, given a matching $\mathsf{nodeMap}$. The resulting transformed graph is denoted H.

1. Extend $\mathsf{nodeMap}$ to a total function $\mathsf{nodeMap}_1$ from $\mathsf{Lhs.Nodes} \cup \mathsf{Rhs.Nodes}$ by adding fresh images (not in $\mathsf{G.Nodes}$) for all elements of $\mathsf{New.Nodes}$;
2. Construct a graph $\mathsf{K} = (\mathsf{nodeMap}_1(\mathsf{Rhs.Nodes}), \mathsf{nodeMap}_1(\mathsf{Rhs.Edges}))$;

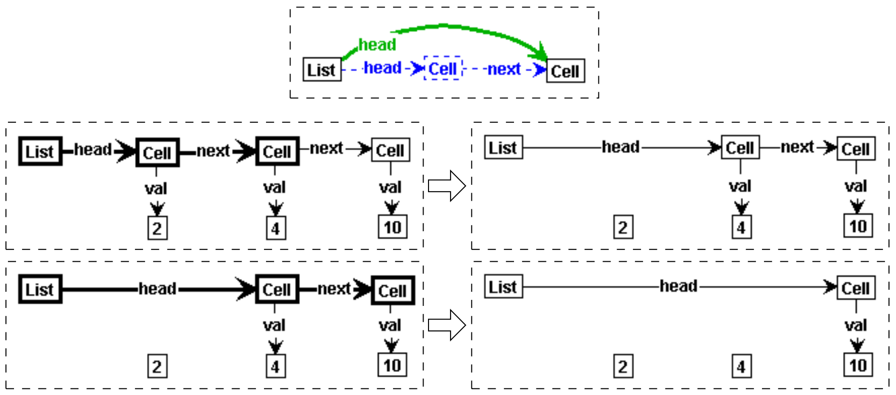

Figure 3: Rule for deleting the head element of a list, with two applications

3. Construct H such that

$$\begin{aligned}\mathsf{H.Nodes} &= \mathsf{K.Nodes} \setminus \mathsf{nodeMap(Del.Nodes)} \\ \mathsf{H.Edges} &= (\mathsf{K.Edges} \setminus \mathsf{nodeMap(Del.Edges)}) \\ &\quad \cap \mathsf{K.src}^{-1}(\mathsf{H.Nodes}) \cap \mathsf{K.tgt}^{-1}(\mathsf{H.Nodes}) \ .\end{aligned}$$

It should be noted that the complications identified above (conflicts between preservation and deletion, and dangling edges) may also be resolved in a different manner, namely by strengthening the conditions under which a rule is considered to be applicable (see Def. 4) so that these cases are automatically excluded. In fact, the solution presented above is the one followed in the so-called *single-pushout approach* whereas the *double-pushout approach* strengthens the application condition instead — see [6, 12] for a thorough discussion. The latter has the advantage that the graphs K and H in the construction above always coincide; in other words, the construction of the transformed graph becomes easier.

Example 6 *Fig. 3 shows a more useful rule than the one in Fig. 2: it deletes the first cell in a list, provided this is not the last one. The figure also shows an application of this rule to the list graph of Fig. 1. On the left, the image of the matching in the host graph is emphasised. In this case, since* $\mathsf{New.Nodes}$ *is empty, the extended mapping* $\mathsf{nodeMap}_1$ *used in the construction above equals* $\mathsf{nodeMap}$*; consequently, the* K *in the construction equals the host graph with one additional* head*-edge. To obtain the target graph, the first* Cell*-node with its incident edges (including the dangling* val*-edge, which is not explicitly mentioned in the rule) is removed from* K*. In the resulting graph, the same rule is applicable once more, resulting in the second transformation in the figure. In the final graph, now, the rule is no longer applicable since no matching from the left hand side exists: the first cell in the list no longer has a successor. Below (Ex. 10) we will show how to define a second rule that takes care of removing the last remaining cell, while making sure that this is only applicable when appropriate.*

2 Graph Transformation Is Logical

The existence of a matching of a graph G in another graph H can be interpreted as a condition on H, namely that the structure described by G can be found somewhere in it. This is for instance used in the applicability condition we have defined in Def. 4: in order to apply a rule, a matching from its left hand side in the host graph must exist. Unfortunately, the kind of properties we can express in this way is limited: essentially, they can only state the *existence* of nodes and edges.

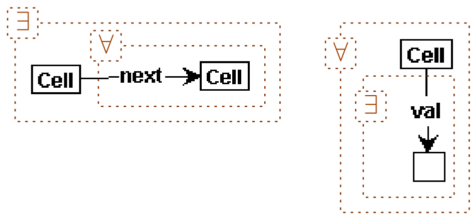

Figure 4: The graph properties of Ex. 7 as nested graphs

Example 7 *Here are some examples of useful properties that are not existential, in the sense above:*

- *Negative conditions: for instance, the property that a given* Cell*-node does* not *have an outgoing* next*-edge. This would be useful to define a rule, complementary to the one in Fig. 3, that deletes the only remaining* Cell*-node from a list.*
- *Invariants: for instance, the property that all* Cell*-nodes have an outgoing* val*-edge. This is a typical property that one would want to verify on a given system.*

A much more powerful way of specifying graph properties, still based on the same principle of graph matching, is obtained by *nesting* the graphs to be matched inside each other and alternating existential and universal quantors for each level of nesting. Two examples, expressing precisely the properties of Ex. 7, are shown in Fig. 4 below.[1]

Symbolically, such nested structures can be understood as existential or universal formulae, exist or univ, respectively, in the following grammar (I and J are finite sets and P, Q_i for $i \in I$ and R_j for $j \in J$ are “property graphs”):

$$\begin{aligned} \mathsf{exist}(\mathsf{P}) &::= \textstyle\bigvee_{i\in I} \exists \mathsf{Q}_i\ \mathsf{univ}_i(\mathsf{Q}_i) \quad (\mathsf{P} \subseteq \mathsf{Q}_i) \\ \mathsf{univ}(\mathsf{P}) &::= \textstyle\bigwedge_{j\in J} \forall \mathsf{R}_j\ \mathsf{exist}_j(\mathsf{R}_j) \quad (\mathsf{P} \subseteq \mathsf{R}_j) \ . \end{aligned}$$

This grammar should be read as follows. Each formula form (either exist or univ) is parameterised with a property graph P, which corresponds to the structure that has already been matched in some outer formula (i.e., within which form is a sub-formula). Each next level quantifies over a super-graph of P, meaning that it requires the existence of additional structure in the host graph (in the case of exist) or states a universal property over all instances of such additional structure (in the case of univ). The semantics is defined by the following rules, where $\mathsf{m}\colon \mathsf{P} \to \mathsf{G}$ stands for the matching already established for the parameter graph P in the host graph G:

$$\begin{aligned} \mathsf{m} \models \mathsf{exist}(\mathsf{P}) \ &:\Leftrightarrow \ \text{there are } i \in I, \mathsf{n}\colon \mathsf{Q}_i \to \mathsf{G} \text{ such that } \mathsf{m} \subseteq \mathsf{n} \text{ and } \mathsf{n} \models \mathsf{univ}_i(\mathsf{Q}_i) \\ \mathsf{m} \models \mathsf{univ}(\mathsf{P}) \ &:\Leftrightarrow \ \text{for all } j \in J, \mathsf{n}\colon \mathsf{R}_j \to \mathsf{G},\ \text{if } \mathsf{m} \subseteq \mathsf{n} \text{ then } \mathsf{n} \models \mathsf{exist}_j(\mathsf{R}_j) \ . \end{aligned}$$

(The notation $\mathsf{m} \subseteq \mathsf{n}$, where $\mathsf{m}\colon \mathsf{P} \to \mathsf{G}$ and $\mathsf{n}\colon \mathsf{Q} \to \mathsf{G}$ with $\mathsf{P} \subseteq \mathsf{Q}$, means that we can create n from m by adding images (in G) for the nodes in $\mathsf{Q.Nodes} \setminus \mathsf{P.Nodes}$.) It should be remarked that the sets I and J may be empty: $\bigvee \emptyset$ is logically equivalent to $\mathbf{ff}$ and $\bigwedge \emptyset$ to $\mathbf{tt}$. As a consequence, a sub-formula $\forall \mathsf{R}_j\ \mathsf{exist}_j(\mathsf{R}_j)$ where $\mathsf{exist}_j = \bigvee \emptyset$ actually expresses the *non-existence* of the structure R_j in the host graph. An example of this is found in the left hand side condition of Fig. 4: the inner, ∀-quantified structure is in fact forbidden.

The rules above define under what circumstances a formula is satisfied by a *matching*. In the end, however, we want to use such formulae to specify properties of *graphs*. For this purpose, we use the one-to-one correspondence between graphs and matchings of the *empty property graph*, here denoted $\emptyset$, into it: that is, we equate a host graph G with (the

[1]This representation as nested graphs is essentially that of *existential graphs* as studied in, e.g., [20, 26, 7]. We have worked out an alternative, more elegant but much more complicated, representation as tree-shaped diagrams in the category of graphs in [18].

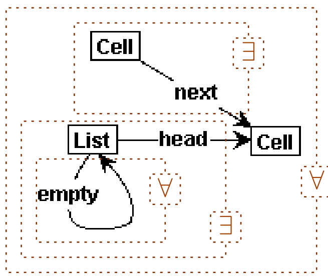

Figure 5: Nested graph property with an existential disjunction

unique) matching $\mathsf{m}\colon\emptyset\to\mathsf{G}$. To be precise, we define, for arbitrary formulae $\mathsf{form}(\emptyset)$ (either $\mathsf{exist}(\emptyset)$ or $\mathsf{univ}(\emptyset)$):

$$\mathsf{G}\models\mathsf{form}\quad :\Leftrightarrow\quad \mathsf{m}\colon\emptyset\to\mathsf{G}\models\mathsf{form}\ .$$

Example 8 *Another, more involved example of a nested graph property is given in Fig. 5. This expresses that every cell either has a predecessor or is the head element of a list, in which case, moreover, the list does not have an* empty*-self-edge. The graph of Fig. 1 satisfies this property: it allows three matchings of the outermost (universally quantified) graph, one of which can be extended to a matching of the bottom (existentially quantified) inner graph, where, moreover, the innermost (universally quantified) graph, consisting of the* empty*-edge, is absent; the other two outermost matchings can be extended to matchings of the top (existentially quantified) inner graph.*

Clearly, with these nested graphs we can express much stronger properties than with single matchings. The following is a consequence of the work on existential graphs, and also follows from [18]:

Theorem 9 *Nested graphs, interpreted as graph properties in the way defined above, are as expressive as first order logic with binary relations (but without equality).*

The restriction to logic without equality is actually quite unfortunate, since it prevents us from doing any kind of *counting*. There are many useful properties that involve, for instance, the uniqueness of nodes with a certain property; for instance, the fact that (in our list example) Cell-nodes are unshared, i.e., have at most one predecessor. It follows from Th. 9 that this and similar properties cannot be expressed with the nested graphs presented above. There are several possible ways to lift this restriction.

- Limit the matchings allowed in Def. 4 to injective ones only. This is a definition often seen in the graph transformation literature: although it may cost a (in the worst case super-exponential) blowup in the number of rules required to model a particular behaviour, in many practical examples it seems quite reasonable.
- Extend the subgraph relation between P and the Q_i and R_j in the grammar of exist and univ to (non-injective) matchings $\mathsf{m}_i\colon\mathsf{P}\to\mathsf{Q}_i$ resp. $\mathsf{n}_j\colon\mathsf{P}\to\mathsf{R}_j$. This is the solution presented in [18], at the cost of additional technical complexity. A consequence is that the resulting structure cannot be depicted as a nested graph any more.
- Introduce special edges in the property graphs that stand for the equality of nodes. We present this solution below.

Negative application conditions. The idea of using some form of nesting to enhance the control over rule applications is far from new: it was described first in [14] for a single level of nesting (resulting in so-called negative application conditions) and extended in [15]

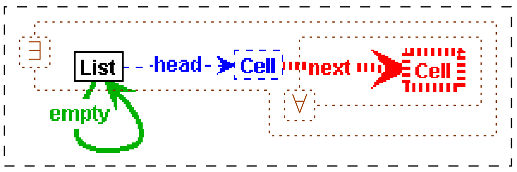

Figure 6: Deletion of the last remaining element from a list

to a second level of nesting (so-called conditional application conditions). The idea is to regard the LHS Lhs of a rule $\langle\mathsf{Lhs},\mathsf{Rhs}\rangle$ as the starting point of a formula $\exists\mathsf{Lhs}\;\mathsf{univ}(\mathsf{Lhs})$. Any matching that is discovered to satisfy the sub-formula $\mathsf{univ}(\mathsf{Lhs})$ gives rise to a rule application.

Example 10 *Fig. 6 shows a production rule for deleting the head element of a list in case this is also the final element, i.e., if it has no outgoing* next*-edge. This involves a negative application condition, here depicted as a nested universal sub-graph (without further sub-conditions, so that the universal property works as a negative condition). As a consequence, this rule is not applicable in the list graph of Fig. 1, but it is applicable to the final (bottom right hand) graph of Fig. 3.*

In addition to the explicit bounding box for the negative condition, in Fig. 6 we have also distinguished the relevant subgraph by using very wide, closely dashed (red) lines. In future examples of rules with negative application conditions we will actually leave out the bounding boxes: each connected sub-graph drawn in these wide, closely dashed lines implicitly stands for a universal sub-formula without further sub-formulae, which is therefore works as a negative condition.

Equality and regular expressions. So far we have used ordinary graph morphisms as matchings. We now strengthen the type of properties we can express by using a different class of labels in P, which should not be matched by single edges of G but rather by (possibly empty) paths through G. We will use $\mathsf{H.Paths}$ to denote the set of such paths. Formally:

$$\begin{aligned}\mathsf{H.Paths} = \{\mathsf{v}_1{\cdot}\mathsf{a}_1{\cdot}\mathsf{v}_2\cdots\mathsf{a}_{n-1}{\cdot}\mathsf{v}_n \in \mathsf{H.Nodes}{\cdot}(\mathsf{Labels}{\cdot}\mathsf{H.Nodes})^* \mid\\ \forall 1 \le i < n : (\mathsf{v}_i,\mathsf{a}_i,\mathsf{v}_{i+1}) \in \mathsf{H.Edges}\}\enspace .\end{aligned}$$

If $\mathsf{p} \in \mathsf{G.Paths}$ for some graph G, we use $\mathsf{lab}(\mathsf{p})$ to stand for the sequence of labels in p, $\mathsf{src}(\mathsf{p})$ for the first node and $\mathsf{tgt}(\mathsf{p})$ for the last node in p; hence, if $\mathsf{p} = \mathsf{v}_1{\cdot}\mathsf{a}_1{\cdot}\mathsf{v}_2\cdots\mathsf{a}_{n-1}{\cdot}\mathsf{v}_n$ then $\mathsf{src}(\mathsf{p}) = \mathsf{v}_1$, $\mathsf{lab}(\mathsf{p}) = \mathsf{a}_1\cdots\mathsf{a}_{n-1}$ and $\mathsf{tgt}(\mathsf{p}) = \mathsf{v}_n$.

Definition 11 (path expression language) *A* path expression language *over* Labels *is a pair* $\langle\mathsf{Exprs},\mathsf{sat}\rangle$*, where* Exprs *is a set of path expressions with* $\mathsf{Labels} \subseteq \mathsf{Exprs}$*, and* $\mathsf{sat}\subseteq \mathsf{Labels}^* \times \mathsf{Exprs}$ *a* satisfaction relation *between sequences of labels in* Labels *and path expressions, such that for all* $\mathsf{a} \in \mathsf{Labels}$*:*

$$\mathsf{s}\ \mathsf{sat}\ \mathsf{a} \iff \mathsf{s} = \mathsf{a}\enspace .$$

If $(\mathsf{s},\mathsf{x}) \in\mathsf{sat}$ for some label sequence $\mathsf{s} \in \mathsf{Labels}^*$ and path expression $\mathsf{x} \in \mathsf{Exprs}$ we say that s *satisfies* x; we usually denote this in infix notation, as $\mathsf{s}\ \mathsf{sat}\ \mathsf{x}$. The definition specifies that a path expression that is actually an element of Labels (recall that $\mathsf{Labels} \subseteq \mathsf{Exprs}$) is satisfied only by itself. A prime example of a path expression language is that of *regular expressions*, RegExprs, generated by the following grammar:

$$\mathsf{x}\ ::=\ =\ \mid\ \mathsf{a}\ \mid\ \mathsf{x}|\mathsf{x}\ \mid\ \mathsf{x}\cdot\mathsf{x}\ \mid\ \mathsf{x}^*\enspace .$$

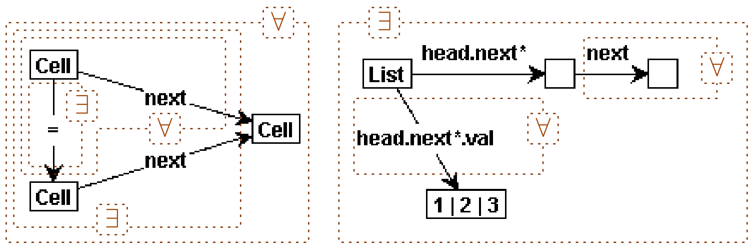

Figure 7: Nested graph properties with path expressions

(where $=$ stands for the empty sequence.) Satisfaction is defined as usual:

$$\begin{aligned}
\mathsf{s\ sat} = \quad &:\Leftrightarrow \quad \mathsf{s} = \varepsilon \\
\mathsf{s\ sat\ a} \quad &:\Leftrightarrow \quad \mathsf{s} = \mathsf{a} \\
\mathsf{s\ sat\ x_1|x_2} \quad &:\Leftrightarrow \quad \mathsf{s_1\ sat\ x_1} \text{ or } \mathsf{s_2\ sat\ x_2} \\
\mathsf{s\ sat\ x_1 \cdot x_2} \quad &:\Leftrightarrow \quad \exists \mathsf{s_1, s_2} \in \mathsf{Labels}^* : \mathsf{s} = \mathsf{s_1 \cdot s_2}, \mathsf{s_1\ sat\ x_1}, \mathsf{s_2\ sat\ x_2} \\
\mathsf{s\ sat\ x^*} \quad &:\Leftrightarrow \quad \exists \mathsf{s_1}, \ldots, \mathsf{s}_n \in \mathsf{Labels}^* : \mathsf{s} = \mathsf{s_1} \cdots \mathsf{s}_n, \forall 1 \leq i \leq n : \mathsf{s}_i \mathsf{\ sat\ x} \ .
\end{aligned}$$

For instance, both ε sat next*, next sat next* and next·next sat $@{-}^*$, and both 1 sat $1|2|3$ and 2 sat $1|2|3$. We generalise the notion of matching to graphs over path expressions as follows:

Definition 12 (path matching) *Given a path expression language* $\langle\mathsf{Exprs}, \mathsf{sat}\rangle$ *over* Labels, *a path matching of a graph* P *over* Exprs *(i.e., where the labels in* P *are actually path expressions) into a graph* G *over* Labels *is a function* nodeMap: P.Nodes $\to$ G.Nodes *such that for all* $(\mathsf{v}, \mathsf{x}, \mathsf{w}) \in$ P.Edges*:*

$$\exists \mathsf{p} \in \mathsf{G.Paths} : \mathsf{src(p)} = \mathsf{nodeMap(v)}, \mathsf{lab(p)\ sat\ x}, \mathsf{tgt(p)} = \mathsf{nodeMap(w)} \ .$$

Example 13 *Fig. 7 shows two nested graph properties. The property on the left hand side is an invariant, expressing that every* Cell*-node has a unique predecessor. Note that the* $=$*-labelled edge specifies that its source and target node are matched by the same node of the host graph, i.e., that the matching is non-injective on these nodes.*

The right hand side is an existential property, expressing that from a List*-node there is a* Cell*-node without successor, and moreover, that no reachable cell node has a* val*-edge to a node with self-edge 1, 2 or 3. For instance, in the initial (top left hand) graph of Fig. 3, this property is not fulfilled because the 2-labelled node still connected to the list; but once we have deleted the first* Cell*-node (bottom left) the property holds, as indicated by the emphasised part of the graph in Fig. 7.*

It should be clear from this example that path expressions enhance the expressive power of nested graph properties. On the one hand we can now state properties about node equality, using $=$-labelled edges; hence we have gained (among other things) the ability to count nodes. On the other hand, the Kleene star enables us to specify paths of arbitrary length, which implies that we can state properties about transitive closure and connectivity; this takes us outside standard first-order logic. Thus, we have the following result (compare Th. 9):

Theorem 14 *Nested graphs with path expressions, interpreted as graph properties in the way defined above, are properly more expressive than first order logic with binary relations and equality.*

In order to use this expressive power in graph transformations, we extend the definition of a rule (Def. 4) so that Lhs and Rhs are graphs over Exprs rather than Labels. However, this only makes sense for that part of the rule that controls the applicability: it is not clear

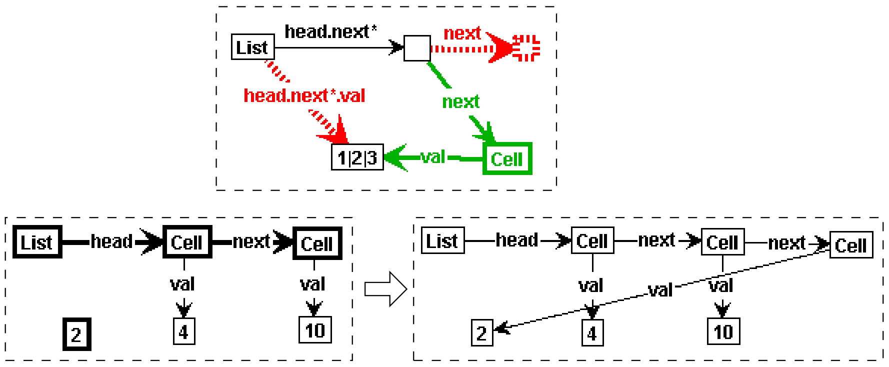

Figure 8: Production rule with path expressions and an application

what it would mean if Del.Edg or New.Edges are Exprs-labelled. Instead, for the edges scheduled to be added or deleted we want to use

Definition 15 (production rule with path expressions) *A production rule with path expressions is a tuple* $\langle \mathsf{Lhs}, \mathsf{Rhs} \rangle$ *of graphs over* Exprs*, where* $\mathsf{lab}(e) \in \mathsf{Labels}$ *for all* $e \in \mathsf{Del.Edges} \cup \mathsf{New.Edges}$*. The rule is* applicable *to a graph* G *if there is a path matching of* Lhs *into* G*.*

The application of a production rule with path expressions is the same as for ordinary production rules, thanks to the fact that the deleted and created nodes and edges take their labels from Labels.

Example 16 *Fig. 8 shows a production rule, essentially consisting of the nested graph property of Fig. 7 as its left hand side. The right hand side specifies that the unconnected node identified in the LHS should be appended to the list.*

3 Graph Transformation Is Difficult

We have advertised graph transformation as a technique that is easy to grasp, powerful and (in certain application domains) natural. Why, then, is this technique not part of the tool kit of more formal methods researchers?

We believe that part of the answer to this question is: because the area of graph transformation is perceived to be difficult, and to be centred on questions of a very theoretical nature. And there is certainly some justification for this feeling. A sizeable fragment of the literature on graph transformation treats the subject on a very abstract level, in a categorical setting; for those not familiar or even averse to category theory, this can be a great barrier to appreciating the joys of graph transformation. In this perspective, it is almost certainly a mistake in public relations to use terms like “Single Pushout Approach” to refer to an intuitively straightforward technique: it suggests that, in order to apply the technique, one has to know what a pushout is, whereas we believe that this is completely irrelevant in most cases.

This is not to deny that category theory is a marvellous way to abstract away from differences in the particular kinds of graphs that one is transforming (typed or untyped, with binary or hyperedges, flat or hierarchical, with or without attributes, to name some choices), and there is much to be gained from lifting some of the general issues involved in any kind of transformation (or rewriting) to this abstract level. For those interested in such theoretical issues we briefly discuss some of them.

Independence of transformations (see, e.g., [6, 11]). This refers to the question if two transformations have overlapping or conflicting effects — for instance, because one transformation removes a node or edge that the other requires in order to be applicable. If such conflicts do not occur, the transformations are called *independent*. This may have important consequences: For instance, in the context of system verification, it may be unnecessary to compute all transformations even in a full state space exploration. In *op. cit.* this problem is studied on the level of properties of the categories involved, so that it does not have to be done again for all different types of graphs.

Constraints and conditions on graphs and graph transformations (sse, e.g., [14, 15, 9, 24]). In Sect. 2 we have shown a fairly straightforward way to formulate properties of our graphs, but again, one would prefer not to re-invent the wheel for each particular type of graph. There are two places where properties of graphs play a special role in the framework of graph transformation: as *invariants* that one would like to hold on all graphs, or as *application conditions* that control the applicability of rules. In *op. cit.* these types of properties and their interrelation are studied.

Compositionality of graph production systems (see, e.g., [13]). In process algebra terminology, graph transformations define a *reduction semantics*: each state "reduces" to the next without interference from the environment — in other words, graph transformations define a "closed world semantics" in which all components of the system being modelled have to be included in the model. In *op. cit.* it is studied how the transformation framework can be enriched with contextual information, so that the behaviour of a complete system can instead be modelled on the level of individual components, which can be put together afterwards.

4 Graph Transformation Wants You!

Although the field of graph transformation stems from the beginning of the 70s and so is, to computer science standards, downright venerable, its application to behavioural modelling and in particular behavioural verification is relatively young. In this contribution we hope to have given you a taste of how it works and an impression of how it could be applied in that area, but there are many open issues. We list a few of them here, with some references; we believe all of these areas to be worthy of further investigation.

Specification. In Sect. 1 we have called graph transformations a natural technique for the specification of semantics for dynamic changes, on particular in object-oriented systems. One paper in which this has been worked out in concrete detail for a relatively large fragment of Java is [5]; another approach is taken in [8], who (essentially) define a special object-based language with a graph transformation semantics. However, we are yet far removed from a discipline in which the definition of such a semantics can be easily and systematically written down — essentially an EBNF standard for behavioural semantics.

Verification. Starting with a graph production system, one can generate the corresponding transition system, essentially by computing all rule applications recursively. This opens up the way for extending existing methods for test generation or model checking to graph transformation-based systems. Some studies can be found in [8, 19, 25]; there are, however, major open problems in dealing with the dynamic nature of the states (which cannot be captured by a fixed state vector). Another interesting option is to extend assertional reasoning to graph production rules: the theory of graph properties mentioned in Sect. 3 lays down at least the terminology in which this problem may be addressed, but the connection to predicate transformation yet has to be tackled. Yet another option is to regard a graph production system as an extended Petri net and transfer techniques from that area, as seen, e.g., in [1, 2].

Abstraction. In the context of the verification issue, we would especially like to address abstraction, as an approximative technique. We believe that graphs offer a very clean setting to study state abstractions; this brings us in the realm of static *shape* analysis á la Sagiv et al. [22, 23] and abstract interpretation. A first proposal was described in [16]. Abstraction of a different kind, using principles from Petri net unfoldings, have also been proposed in [3, 4].

Compositionality. In Sect. 3 we have briefly referred to the general theory for compositionality that has recently been developed. However, this work takes the reduction semantics as a starting point and derives contextual rules from that. We believe that, instead, it can be quite natural to start off with transformation rules that explicitly take context into account, and whose effect may include the sending and receiving of (sub)graphs. For instance, in the context of the running example on lists (e.g., Ex. 16), one may imagine a rule that appends an object "received" from some other component of the system; the identity of the object may be communicated through some parameterisation mechanism in the style of Structural Operational Semantics. As far as we are aware, this combination of ideas from graph transformation and process algebra has not been considered at all so far.

As a consequence of the predominance of category theory in much of the graph literature research, it is also obligatory to place this paper within the categorical framework. What we have defined, on a set-theoretic level, is an instance of the single-pushout approach, which cannot be formulated easily in the double pushout approach — the reason essentially being the fact that we do not allow so-called *parallel edges*, or in other words, that our edges do not have an identity of their own. From the point of view of the algebraic framework, this is a real drawback: much of the theory referred to above applies only in the double pushout approach. The reason why we have nevertheless set up our definitions as we did is twofold: it makes for an easier technical presentation, but more importantly, we like to equate edges to binary relations. In that light it does not make sense for edges to have an identity themselves.

References

[1] P. Baldan, A. Corradini, and B. König. A static analysis technique for graph transformation systems. In K. Larsen and M. Nielsen, editors, *Concur 2001: Concurrency Theory*, volume 2154 of *LNCS*, pages 381–395. Springer-Verlag, 2001.

[2] P. Baldan, A. Corradini, and B. König. Verifying finite-state graph grammars: An unfolding-based approach. In P. Gardner and N. Yoshida, editors, *Concurrency Theory (CONCUR)*, volume 3170 of *LNCS*, pages 83–98. Springer-Verlag, 2004.

[3] P. Baldan, B. König, and B. König. A logic for analyzing abstractions of graph transformation systems. In R. Cousot, editor, *Static Analysis*, volume 2694 of *LNCS*, pages 255–272. Springer-Verlag, 2003.

[4] P. Baldan, B. König, and I. Stürmer. Generating test cases for code generators by unfolding graph transformation systems. In Ehrig et al. [10], pages 194–209.

[5] A. Corradini, F. L. Dotti, L. Foss, and L. Ribeiro. Translating java into graph transformation systems. In Ehrig et al. [10], pages 383–389.

[6] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. Algebraic approaches to graph transformation, part I: Basic concepts and double pushout approach. In Rozenberg [21], chapter 3, pages 163–246.

[7] F. Dau. *The Logic System of Concept Graphs with Negation*, volume 2892 of *LNCS*. Springer-Verlag, 2003.

[8] F. L. Dotti, L. Foss, L. Ribeiro, and O. M. dos Santos. Verification of distributed object-based systems. In E. Najm, U. Nestmann, and P. Stevens, editors, *Formal Methods for Open Object-based Distributed Systems (FMOODS)*, volume 2884 of *LNCS*, pages 261–275. Springer-Verlag, 2003.

[9] H. Ehrig, K. Ehrig, A. Habel, and K.-H. Pennemann. Constraints and application conditions: From graphs to high-level structures. In Ehrig et al. [10], pages 287–303.

[10] H. Ehrig, G. Engels, F. Parisi-Presicce, and G. Rozenberg, editors. *Second International Conference on Graph Transformation*, volume 3256 of *LNCS*. Springer-Verlag, 2004.

[11] H. Ehrig, A. Habel, J. Padberg, and U. Prange. Adhesive high-level replacement categories and systems. In Ehrig et al. [10], pages 144–160.

[12] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini. Algebraic approaches to graph transformation, part II: Single pushout approach and comparison with double pushout approach. In Rozenberg [21], pages 247–312.

[13] H. Ehrig and B. König. Deriving bisimulation congruences in the dpo approach to graph rewriting. In I. Walukiewicz, editor, *Foundations of Software Science and Computation Structures (FOSSACS)*, volume 2987 of *LNCS*, pages 151–166. Springer-Verlag, 2004.

[14] A. Habel, R. Heckel, and G. Taentzer. Graph grammars with negative application conditions. *Fundamenta Informaticae*, 26(3/4):287–313, 1996.

[15] R. Heckel and A. Wagner. Ensuring consistency of conditional graph grammars – a constructive approach. In A. Corradini and U. Montanari, editors, *Joint COMPUGRAPH/SEMAGRAPH Workshop on Graph Rewriting and Computation*, volume 2 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers B.V., 1995.

[16] A. Rensink. Canonical graph shapes. In D. A. Schmidt, editor, *Programming Languages and Systems — European Symposium on Programming (ESOP)*, volume 2986 of *LNCS*, pages 401–415. Springer-Verlag, 2004.

[17] A. Rensink. The GROOVE simulator: A tool for state space generation. In J. Pfalz, M. Nagl, and B. Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, volume 3062 of *LNCS*, pages 479–485. Springer-Verlag, 2004.

[18] A. Rensink. Representing first-order logic using graphs. In H. Ehrig, G. Engels, F. Parisi-Presicce, and G. Rozenberg, editors, *International Conference on Graph Transformations (ICGT)*, volume 3256 of *LNCS*, pages 319–335. Springer-Verlag, 2004.

[19] A. Rensink, Á. Schmidt, and D. Varró. Model checking graph transformations: A comparison of two approaches. In H. Ehrig, G. Engels, F. Parisi-Presicce, and G. Rozenberg, editors, *International Conference on Graph Transformations (ICGT)*, volume 3256 of *LNCS*, pages 226–241. Springer-Verlag, 2004.

[20] D. D. Roberts. The existential graphs. *Computers and Mathematics with Applications*, 6:639–663, 1992.

[21] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume I: Foundations. World Scientific, Singapore, 1997.

[22] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Trans. Prog. Lang. Syst.*, 20(1):1–50, Jan. 1998.

[23] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Prog. Lang. Syst.*, 24(3):217–298, May 2002.

[24] G. Taentzer and A. Rensink. Ensuring structural constraints in graph-based models with type inheritance. In *Fundamental Approaches to Software Engineering (FASE)*, LNCS. Springer-Verlag, 2005.

[25] D. Varró. Automated formal verification of visual modeling languages by model checking. *Journal of Software and Systems Modelling*, 3(2):85–113, 2004.

[26] M. Wermelinger. Conceptual graphs and first-order logic. In *International Confence on Conceptual Structures*, volume 954 of *LNAI*, pages 323–337. Springer-Verlag, 1995.