Electronic Communications of the EASST
Volume 29 (2010)

Proceedings of the Ninth International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2010)

On A Graph Formalism for Ordered Edges

Maarten de Mol and Arend Rensink

12 pages

Guest Editors: Jochen Küster, Emilio Tuosto
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

On A Graph Formalism for Ordered Edges

Maarten de Mol* **and Arend Rensink**

http://www.cs.utwente.nl/~molm, molm@cs.utwente.nl
http://www.cs.utwente.nl/~rensink, rensink@cs.utwente.nl
Department of Computer Science, University of Twente
Enschede, The Netherlands

Abstract: Though graphs are flexible enough to model any kind of data structure in principle, for some structures this results in a rather large overhead. This is for instance true for *lists*, i.e., edges that are meant to point to an ordered collection of nodes. Such structures are frequently encountered, for instance as ordered associations in UML diagrams. Several options exist to model lists using standard graphs, but all of them need auxiliary structure, and even so their manipulation in graph transformation rules is not trivial.

In this paper we propose to enrich graphs with special ordered edges, which more naturally represent the intended structure, and define how lists can be manipulated. We show that the resulting category satisfies sufficient HLR properties to apply standard algebraic graph transformation. We believe that in a context where lists are common, the cost of a more complicated graph formalism is outweighed by the benefit of a smaller, more appropriate model and more straightforward manipulation.

Keywords: Graph Rewriting, Ordered Edges

1 Introduction

The context of the work in this paper is *graph transformation*. This means that we use graphs, essentially only consisting of nodes and edges, to model different kinds of structures such as real-world systems or software concepts. A rich source of such structures comes from software engineering, in the form of UML models. Graph transformation offers a mathematically well-founded method for systematically encoding changes to graphs; this in turn can be used to describe the dynamics of the system being modelled.

In principle, appropriate compositions of the basic building blocks of nodes and binary edges can encode arbitrary structures. In many cases the resulting graphs reflect the original structures quite naturally. There are, however, situations in which the encoding is awkward, for instance because it requires auxiliary elements in the graph that do not directly reflect anything from the original structure. This impacts the understandability and complexity of the encoding, and thus decreases the usability of graph transformation. In such cases, one may choose to use a richer graph formalism instead, which more closely reflects the structures at hand. Examples of enrichments, introduced exactly for the reason of modelling particular structures more naturally, are: attributed graphs [EEPT06a], hierarchical graphs [DHP02], and hypergraphs [Hab92].

* Supported by the EU Artemis project CHARTER

There is, however, a price to pay for graph enrichments, in the form of added complexity in their usage and understanding (often called the *learning curve*), as well as in their manipulation, both on the level of theory and of implementation. Enrichments in the graph formalism are only justified if the complexity increase is outweighed by the corresponding advantages in modelling.

In this paper we propose an enrichment of the basic graph formalism to cope with the structural concept of *ordered lists*. Such lists occur frequently in practice, for instance in the form of ordered associations in UML diagrams or array- and list-like structures in software. We will argue that encoding lists using simple graphs introduces spurious elements and thus increases their complexity; also the manipulation of the encodings is non-trivial.

In order to justify the cost of a more complex formalism on the level of theory, we show that DPO graph rewriting is well-behaved in the resulting category of list graphs. For a suitably chosen admissible[CL03] subclass of M-morphisms, we prove that pushouts along M-morphisms exist, and that these pushouts are *partial VK squares*[Hei09]. We then make use of [Hei09], in which Heindel proves that these conditions imply the important HLR properties[Pad93]. Note that our category is *not* HLR adhesive [EEPT06b]; see Section 3 for a counter-example.

In the next section, we motivate and explain our extension on an intuitive level, using an example inspired by the Olympic winter games. After that, Section 3 presents the formal definitions and states the main theoretical result. We show the use of list graphs in Section 4. Finally, Section 5 discusses related work and presents conclusions.

2 Motivation

As a motivating example, we use sporting events taking place in the 2010 Olympic winter games. In particular, we concentrate on ice-skating. Before the games, every skating event has a list of participants; the order in the list corresponds to their starting order at the event. For instance, Figure 1 shows three events (1500 m, 5 km and 10 km for men). The 1500 m event has four participants, in the order from top to bottom; the 5 km event has three participants, namely Kramer, Tuitert and Davis (in that order); and the 10 km has an empty list of participants.

Intuitively straightforward operations one may want to perform on such a list are:

- Appending an element when a new participant is enrolled;
- Removing participants convicted of doping abuse;
- After the event, moving the winner to the top of the list.

A more complex operation is list reversal. For instance, if we started with a ranking list (in which the seasonal best skater is at the top), then it needs to be reversed to get the starting order.

2.1 Plain graph encoding

There are several ways to encode such lists using plain graphs, consisting only of nodes and binary edges. We discuss the main issues.

- The core problem is to specify the order of the elements. For this purpose, one can either rely on an implicit ordering, for instance using indices, or introduce an explicit ordering

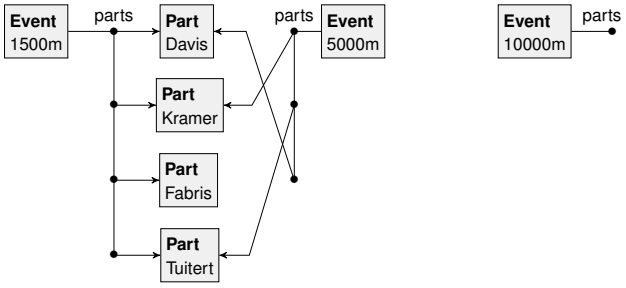


Figure 1: Skating events with overlapping lists of participants

using special edges. Indices require updating whenever elements are added or removed (except at the end of the list).

- Elements can be shared among lists (as Figure 1 shows), or may even occur multiple times in the same list. For this reason, the indices or special edges specifying the ordering cannot be incident to the list elements themselves (this would introduce confusion between the lists); rather, one needs an intermediate layer of "slot" nodes.

- It is often convenient, or even necessary, to express that a given element is in a particular list. To encode this information, we need further special edges pointing from the list owner to the elements, or vice versa.

- Many list operations explicitly refer to the first or last element. To express this, either we need negative application conditions stating that the element has no predecessor, respectively successor; or this information can be captured using special edges — which, however, then have to be maintained while manipulating the list.

- The empty list needs to be represented in some special way, as in that case there are no element or slot nodes to attach information to.

Clearly, such a graph representation is expensive, in the sense of requiring many auxiliary elements; moreover, unless one is careful, the last two issues will require case distinctions in transformation rules. From programming, we know an encoding for lists that copes with most of the issues relatively well (in particular avoiding case distinctions), but is expensive in terms of overhead: namely, a circular linked list consisting of "slot" nodes pointing to the elements and back to the list owner, and a special "head" node without an element, marking the start and the end of the list. Figure 2 shows a plain graph encoding of the structure of Figure 1.

Figure 3 shows an example rule that will result in the winner of an event being moved to the start of the list. The figure shows the left hand side and right hand side of the rule; the connecting morphisms are implicit in the positioning of the nodes. The unlabelled nodes are meant to match any node in the graph; in particular, they may match **Head** or **Slot** nodes. A solution that works for properly typed graphs requires inheritance. Note that this only works under the assumption that non-injective matches are allowed.

An important observation is that *the issues discussed above are exactly those one encounters while programming with lists*. This goes against the idea that graph transformation provides an

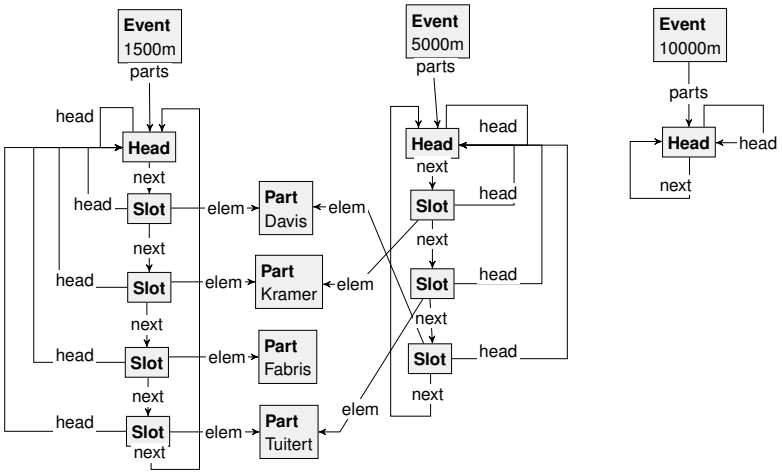


Figure 2: Plain graph representation of the structure in Figure 1

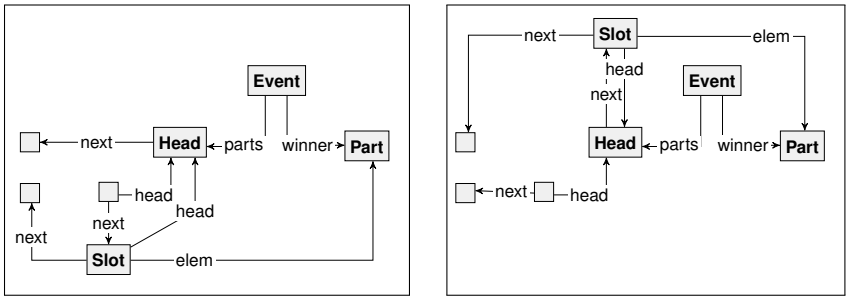


Figure 3: Plain graph rule moving the winner of an event to the top of the list.

abstract, declarative way of manipulating structures. If the graph model is used for the design of a software system, from which an implementation is to be derived, then the graph representation choices will influence the implementation, possibly in unintended ways. For instance, the encoding in Figure 2 makes it unnatural to choose an array-based implementation.

2.2 List edges

The proposal in this paper is to enrich graphs with explicit support for lists, avoiding both the overhead and the "programming" nature of the plain graph encoding. We do this by extending the notion of edges: rather than binary edges with a single source node and a single target node, we propose to use *list edges* of which the target is a sequence of nodes. Thus, list edges are somewhat like hyperedges in that they may have different numbers of tentacles: however, hyperedges typically have a fixed number of tentacles (called the arity) determined by their labels, which is not the case for list edge arity.

For instance, Figure 1 is a straightforward visualisation of a graph with list edges from the **Event** nodes to different sequences of **Part** nodes. The string of "knots" in the edge gives the order of the elements in the list; the arrows from the knots point to the actual elements.

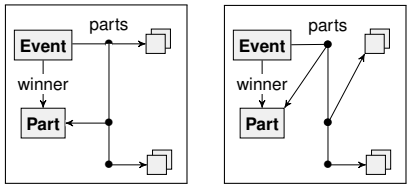


Figure 4: List graph rule moving the winner of an event to the top of the list.

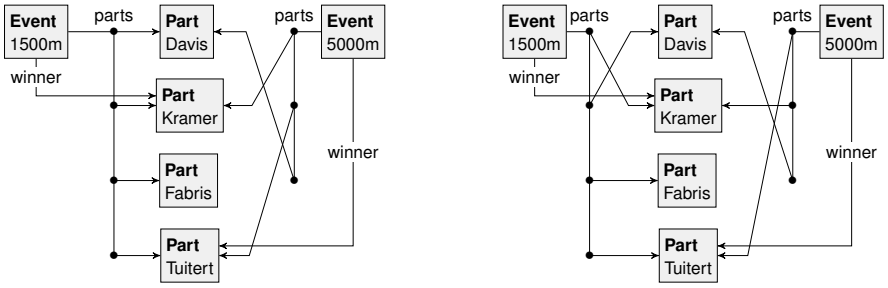


Figure 5: Applying Figure 4 twice to the left hand side graph yields the right hand side graph

The real innovation, however, does not lie in the graphs but in the rules. For these, we introduce a new type of node, called *list nodes*, which will only appear in rules and stand for arbitrary sequences of nodes from the host graph. List nodes can only occur as edge targets, never as sources. Graph morphisms are extended by matching list nodes either to a sequence of plain nodes, or to a single list node. This is extended to list edges in the natural way.

For instance, Figure 4 shows the same rule as Figure 3, but this time for list graphs. The 'doubled' nodes are list nodes. The parts edge in the left hand side matches any list edge in the host graph from an Event node, pointing to an arbitrary sequence of nodes (matched by the upper list node of the LHS), followed by the Part-node that the winner-edge points to, followed by another arbitrary sequence of nodes (matched by the lower list node of the LHS). The effect of the rule is to delete this list edge and create a new one, in which the Part-node and the first sub-sequence are swapped. This has the effect of moving the Part-node to the top of the list.

An example of the application of this rule is shown in Figure 5. The initial state is the same as in Figure 1, but now with Kramer and Tuitert indicated as winners for the 1500m and 5000m respectively. The rule can be applied twice, resulting in the right hand side graph.

3 Formalisation

In this section, we will show that lists can be incorporated in graph theory in a sound manner. For this purpose, we extend a standard representation of multi-sorted graphs with list nodes and list edges. We define an admissible subclass of M-morphisms, and prove that pushouts along M-morphisms exist and are partial VK squares. Using [Hei09], this implies that sufficient HLR properties hold. We will use double pushouts (DPO) for the formalisation of graph rules.

First, we extend a standard (V, E, src, tgt, lab) representation of multi-sorted graphs, by: (1) splitting V into $\hat{V}$ (normal nodes) and $\overline{V}$ (list nodes); and (2) changing the result of tgt from V (a

single node) to $V^\star$ (a sequence of nodes, may be empty). In other words, we add list nodes and replace one-to-one (plain) edges with one-to-many (list) edges:

Definition 1 (*multi-sorted list graphs)*
Let $G = (\hat{V}, \overline{V}, E, src, tgt, lab)$ be a multi-sorted list graph, where:
- $\hat{V}$ and $\overline{V}$ are the sets of plain nodes and list nodes respectively (let V denote $\hat{V} \cup \overline{V}$)
- E is the set of (list) edges
- $\hat{V}$, $\overline{V}$ and E are disjoint
- $src : E \to \hat{V}$ is the function that yields the source node of an edge
- $tgt : E \to V^\star$ is the function that yields the sequence of target nodes of an edge
- $lab : E \to L$ is the labelling function (assuming a fixed set of labels L)

As usual, we will use graph homomorphisms as arrows in our category. A homomorphism $f : G \to H$ is a structure preserving mapping of nodes and edges. In our category, three cases are distinguished: (1) plain nodes are mapped to plain nodes; (2) list nodes are mapped either to list nodes or to sequences of plain nodes; and (3) list edges are mapped to list edges. The one-to-one mapping of list nodes will be used to restrict our graph rules, and the one-to-many mapping of list nodes will be used for the matching of a rule to a graph.

For the sake of convenience, we will combine the mappings of nodes into a single function that always produces a sequence. Furthermore, we will often implicitly convert a singleton sequence to its element or vice-versa; it will always be clear from the context when we do this. Finally, we will write $f_V^\star$ for the sequence homomorphism that is generated by f_V; that is, if f_V is a function from V_G to $V_H^\star$, then $f_V^\star$ is the natural extension that maps $V_G^\star$ to $V_H^\star$.

Definition 2 (*homomorphisms)*
Let $G = (\hat{V}_G, \overline{V}_G, E_G, src_G, tgt_G, lab_G)$ and
$H = (\hat{V}_H, \overline{V}_H, E_H, src_H, tgt_H, lab_H)$ be multi-sorted list graphs.
Let $f = (f_V, f_E)$ with $f_V : V_G \to V_H^\star$ and $f_E : E_G \to E_H$ map the nodes and edges of G to H.
Then, f is a homomorphism when the following conditions hold:
- for all $v_g \in \hat{V}_G$ there exists a $v_h \in \hat{V}_H$ such that $f_V(v_g) = \langle v_h \rangle$
- for all $v_g \in \overline{V}_G$, there either exists a $v_h \in \overline{V}_H$ such that $f_V(v_g) = \langle v_h \rangle$, or $f_V(v_g) \in \hat{V}_H^\star$
- $lab_H \circ f_E = lab_G$
- $src_H \circ f_E = f_V \circ src_G$
- $tgt_H \circ f_E = f_V^\star \circ tgt_G$

The composition of two homomorphisms can now easily be defined by means of a combination of function composition and natural extension to sequences. By construction, it follows that the result is a homomorphism as well, which allows us to define list graphs as a category.

Definition 3 (*composition of homomorphisms)*
If $f = (f_V, f_E) : G \to H$ and $g = (g_V, g_E) : H \to I$ are homomorphisms on list graphs, then $g \circ f$ is defined by $(g_V^\star \circ f_V, g_E \circ f_E)$.

Definition 4 (*list graphs as a category)*
The category $\mathbb{GL}$ consists of list graphs (Definition 1) as objects, homomorphisms (Defini-

tion 2) as arrows and composition as in Definition 3. The identity arrows are the homomorphisms that are pairs of identity functions.

Next, we define a suitable subclass of M-morphisms and show that it is admissible [CL03]. In this paper, we present a part of the proof only; the full proof can be found in [MRH10].

Definition 5 (*M-morphisms in* $\mathbb{GL}$)
A monomorphism $f = (f_V, f_E) : G \to H$ in $\mathbb{GL}$ belongs to the subclass M if for all $v_G \in \overline{V}_G$ there exists a $v_H \in \overline{V}_H$ such that $f_V(v_G) = \langle v_H \rangle$. In other words: a M-morphism does not perform matching of list nodes to sequences, but maps them one-to-one to list nodes only.

Theorem 1 (*M is admissible*)
The subclass M is admissible: M contains the identity morphisms, $\mathbb{GL}$ has pullbacks along M-morphisms and the opposing morphism in the pullback diagram is a M-morphism itself.

Proof (sketch).

- Identity morphisms always map list nodes to themselves, and are therefore M-morphisms.
- Pullbacks are constructed as follows. Suppose that $B \xrightarrow{b} A \xleftarrow{c} C$, and that b is a M-morphism. Let A_B be the subgraph of A that is formed by the image of b. Because b is a M-morphism, B is isomorphic to A_B. Construct the largest subgraph $D \subseteq C$ such that c maps all elements of D to elements of A_B. Then, D is the pullback of $B \xrightarrow{b} A \xleftarrow{c} C$, with $D \to C$ by means of id_D and $D \to B$ by means of $z \circ c$, where z is the isomorphism between A_B and B.
- The opposing morphism is id_D, which is a M-morphism.

Next, we show that $\mathbb{GL}$ also has pushouts along M-morphisms. Again, we we present a part of the proof only; the full proof can be found in [MRH10].

Theorem 2 (*pushouts*)
$\mathbb{GL}$ *has pushouts along M-morphisms.*

Proof (sketch).

- Pushouts are constructed as follows. Suppose that $B \xleftarrow{b} A \xrightarrow{c} C$, and that b is a M-morphism. Assume that B and C are disjoint (if not, find isomorphic graphs that are disjoint). Let B_A be the subgraph of B that are in the image of b. Because b is an M-morphism, A is isomorphic to B_A. Then, $D = C \cup (B \backslash B_A)$ is the pushout of $B \xleftarrow{b} A \xrightarrow{c} C$, with $C \to D$ by means id_C and $B \to D$ by means of $id_{B \backslash B_A} \cup (c \circ z)$, where z is the isomorphism between B_A and A. Note that when edges are added by b (i.e. they appear in $B \backslash B_A$), then the sources and targets of these edges have to be transformed by means of $id_{B \backslash B_A} \cup (c \circ z)$ as well.
- Note that the opposing morphism is again an identity (id_C), and is therefore a M-morphism.

The next step is to show that the constructed pushouts form *partial VK squares* [Hei09]. This is a more involved proof, for which we refer to the technical report [MRH10] completely. Here, we present the definition of partial VK squares only:

Definition 6 (*partial VK squares*)
A pushout ${}^{B \leftarrow A}_{D \leftarrow C}$ is a *partial Van Kampen square* if for each commutative cube on top of the pushout as shown in Figure 6 on the left, which has pullback as back faces such that both b

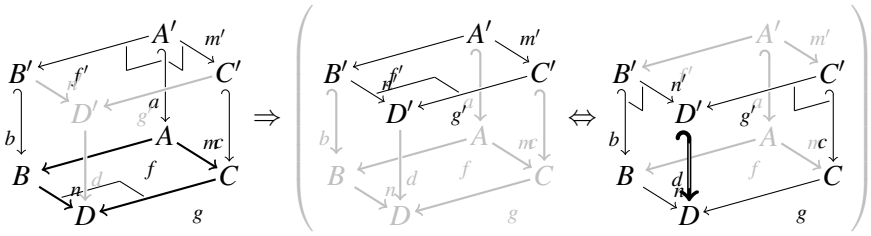


Figure 6: Partial Van Kampen square property

and c are M-morphisms, its top face is a pushout if and only if the front faces are pullback and the morphism d is an M-morphism (as illustrated in Figure 6 on the right).

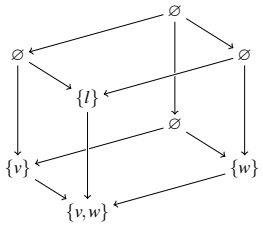
Theorem 3 (*pushouts are partial VK squares)*
The pushouts in $\mathbb{GL}$ are partial VK squares.
Proof: see technical report [MRH10].

In [Hei09], Heindel has shown that the important HLR properties hold in a category with admissible M-morphisms, pushouts along M-morphisms and partial VK squares. Therefore, the combination of Theorems 1, 2 and 3 ensures that graph rewriting is well-behaved in our category $\mathbb{GL}$, using the following standard definition of double pushout (DPO) rewriting:

Definition 7 (*double pushout rewriting)*
A graph production $L \xleftarrow{l} K \xrightarrow{r} R$ is applied to a host graph G with the following procedure:
- Find a morphism m that maps L to G, and a morphism k that maps K to D such that the pushout of $K \xrightarrow{l} L$ and $K \xrightarrow{k} D$ is G (with m).
- Then, build the pushout of $K \xrightarrow{r} R$ and $K \xrightarrow{k} D$, which is the result of applying the rule.

If either of the morphisms m or k does not exist, the rule cannot be applied. The well-behavedness shown above ensures that k is unique (if it exists).

Contrary to our earlier beliefs, $\mathbb{GL}$ is not HLR adhesive [EEPT06b]. This is illustrated by the cube on the right, in which v and w are plain nodes and l is a list node (and no edges occur). The arrows are inclusions, except the one that maps l to $\langle v, w \rangle$. The bottom face is a pushout, the back faces are pullbacks, but the top face is not a pushout. The cube is therefore an example of a pushout that is *not* a VK square.



Unfortunately, the current definitions, although sound, still give rise to some strange behaviour. Suppose that $p = (L \leftarrow K \rightarrow R)$ is a production. Then:

- If R contains list nodes that have no counterpart in K, then the application of p introduces list nodes in the host graph. This is undesirable, because a list node in a normal graph has no meaning; a list node only makes sense in a rule.
- Conversely, if L contains list nodes that have no counterpart in K, then p can never be applied to graphs that do not contain list nodes. This is due to the pushout construction (see Theorem 2), which copies $L \backslash K$ in the host graph.

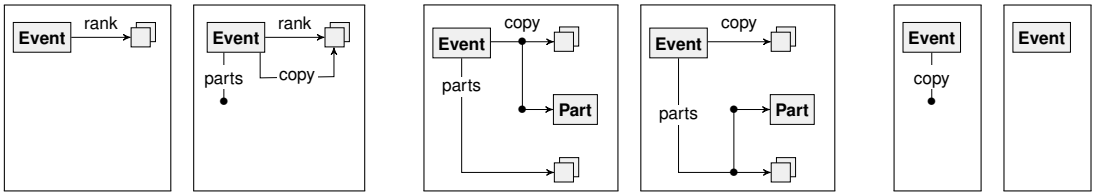


Figure 7: List graph rules creating a reversed parts list out of a rank list.

We will disallow this strange behaviour by demanding that both the morphisms in a production must be surjective with respect to list nodes, which ensures that L and R cannot contain list nodes that do not have a counterpart in K.

Definition 8 (*surjective M-morphisms*)
A M-morphism $f = (f_V, f_E) : G \to H$ in $\mathbb{GL}$ is surjective if for all $v_H \in \overline{V}_H$ there exists a $v_G \in \overline{V}_G$ such that $f_V(v_G) = \langle v_H \rangle$.

Definition 9 (*productions in $\mathbb{GL}$*)
For graph rewriting in the category $\mathbb{GL}$, only productions $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ are allowed in which both l and r are surjective M-morphisms.

It turns out that l and r being surjective is not only a necessary, but even a sufficient condition for ensuring that rules do not introduce list nodes. A proof of this property can again be found in the technical report [MRH10]. This implies that graph rewriting in our category $\mathbb{GL}$ always transforms normal graphs (i.e. without list nodes) to normal graphs.

4 List reversal

We show some more applications of list graph transformations, inspired by the setting of Section 2. In particular, we show how we can obtain a participants list, parts, from a ranking list, rank, by copying and reversing the list. The entire behaviour is specified by the rules in Figure 7.

- The *start* rule copies the rank list into a copy list, and creates an empty parts list. Note that this is a "shallow" copy: the elements are not copied but shared among the lists.
- The *build* rule repeatedly removes the last element from the copy list and appends it to the parts list. By applying this rule as long as possible, eventually the copy list will be empty, at which point the parts list contains all the elements of the original copy list, and hence of the rank list, in reverse order.
- The *finish* rule deletes the empty copy list, completing the reversal process. Note that this rule is only applicable if the copy list is indeed empty.

Figure 8 shows a sequence of applications of these rules.

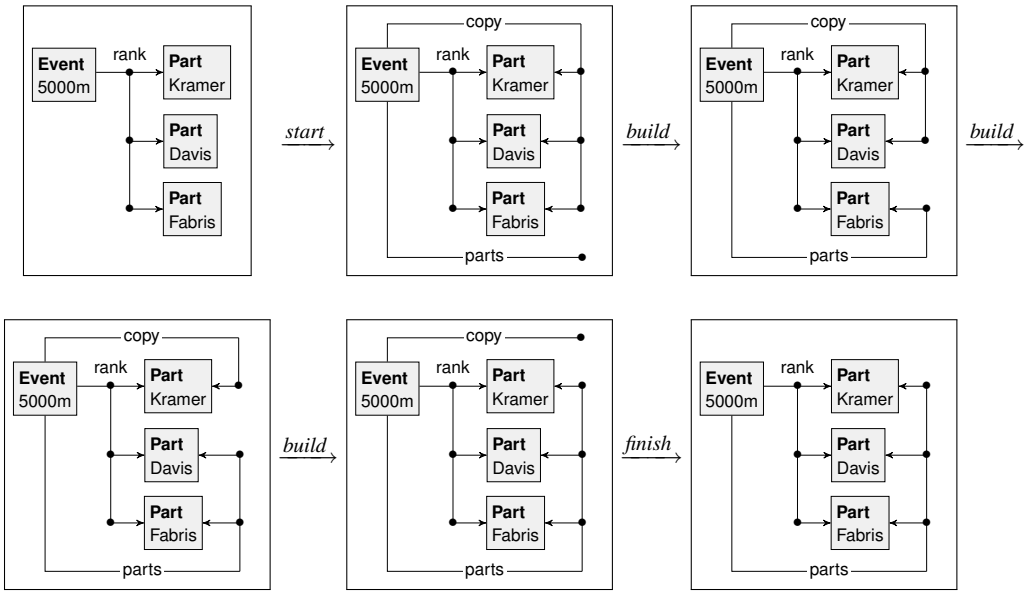


Figure 8: Example production sequence for the rules in Figure 7.

5 Conclusion

In this section, we look back on what we have achieved, and list the good and bad points. We also briefly discuss related work and future extensions.

5.1 Evaluation

We have defined list graphs in order to directly capture ordered structures. We have shown that encoding such structures into plain graphs is awkward and, worse, introduces programming-like structures that break the inherent abstraction of graph-based models. In contrast, the construction and manipulation of list graphs is much more abstract and results in smaller, more intuitive graphs and rules. We have shown that list graphs fit into the theory of algebraic graph rewriting, and so the cost of the more complex graph formalism is low, at least on the level of theory.

On the downside, the way lists are manipulated on the theoretical level is not attractive from an implementation point of view. List edges are deleted and created as a whole, which, when taken literally, would mean that entire lists are discarded and constructed every time a single element is added or deleted. An implementation should instead recognise and efficiently deal with frequently occurring patterns of list usage. A first attempt is to identify re-use of list edges with a static analysis of stable nodes and edges, but it is yet unclear how this can be generalised.

It may be remarked that our lists break the usual symmetrical treatment of edge sources and targets, since list nodes may only occur at an edge target. In this regard, we have been led by the intended application of the enriched formalism. From the theoretical perspective there is no reason to forbid list nodes at edge sources: our theory smoothly extends to standard hyperedges (keeping our special notion of morphism), which do not have a distinguished source node at all.

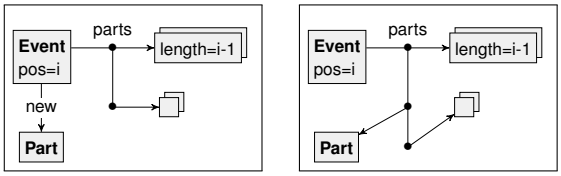


Figure 9: List graph rule inserting an element at a specified position.

5.2 Related work

As far as we have been able to determine, there is essentially no prior work on enriching the basic graph formalism with lists. On a more pragmatic level, however, many tools offer ways to deal with ordered structures or associations, if only by suggesting a default encoding or syntactic sugar. For instance, FUJABA reflects programming structures such as lists and arrays into the rules, and provide notations to traverse them conveniently (see [MZ04]). FUJABA's handling of ordered edges is formalised in [Zün01]. For VIATRA2 it is suggested in [VB07] to use relations over relations to encode ordering. In general it is difficult to find information about such pragmatic solutions.

Remotely related are extensions to deal with parallel or amalgamated rule applications (e.g., [Tae97]), since in this setting the rules also have nodes that can be mapped to more than one graph node (a prime instance are the *set nodes* of PROGRES, see [Sch97]). However, the connection stops there: the purpose and technical contribution of this work is entirely different.

5.3 Future work

So far, the concepts in this paper only exist in theory. The proof of their usability can only come through an implementation. The natural way to go is to extend our research vehicle GROOVE (see [Ren04]) to list graphs. However, this will require a major refactoring to generalise to hyperedges — quite apart from the fact that GROOVE implements SPO and not DPO rewriting.

Instead, we first plan to use these ideas to define a suitable transformation language in the project CHARTER[1], in the context of which this work has been carried out. For this project we will provide a tool that compiles graph transformation systems to Java source code which accesses and manipulates the actual graphs through a predefined API. Since ordered lists and arrays are a common feature in the graphs we will have to deal with, it is imperative to have a suitable, declarative way to specify their transformation.

A theoretical extension that would add quite a bit of power to the formalism, and make it even more generally usable, is indexing. Currently there is no way to specify or reason about the position of an element in a list. We conjecture that this requires only a minor extension, namely to add a default unmodifiable length attribute to all list nodes. Morphisms then have to respect the length of list nodes, in the following way: if a morphism maps a list node to another list node, then the value of the length attribute should remain unchanged, whereas if the image is a sequence of plain nodes, the value of the length attribute should equal the actual length of the sequence. For instance, Figure 9 specifies that a Part-node should be inserted at index i.

[1] See http://charterproject.ning.com/.

Bibliography

[CL03] J. R. B. Cockett, S. Lack. Restriction Categories II: Partial Map Classification. *Theoretical Computer Science* 294(1/2):61–102, 2003.

[DHP02] F. Drewes, B. Hoffmann, D. Plump. Hierarchical Graph Transformation. *J. Comput. Syst. Sci.* 64(2):249–283, 2002.

[EEPT06a] H. Ehrig, K. Ehrig, U. Prange, G. Taentzer. Fundamental Theory for Typed Attributed Graphs and Graph Transformation based on Adhesive HLR Categories. *Fundam. Inform.* 74(1):31–61, 2006.

[EEPT06b] H. Ehrig, K. Ehrig, U. Prange, G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. Springer, 2006.

[Hab92] A. Habel. *Hyperedge Replacement: Grammars and Languages*. Springer-Verlag New York, Inc., 1992.

[Hei09] T. Heindel. *A Category Theoretical Approach to the Concurrent Semantics of Rewriting*. PhD thesis, Universität Duisburg-Essen, 2009.

[MRH10] M. de Mol, A. Rensink, T. Heindel. A Graph Formalism For Ordered Edges. 2010. Technical Report, University of Twente, The Netherlands. To appear. Preliminary version available at http://wwwhome.cs.utwente.nl/~molm/list_techreport.pdf.

[MZ04] T. Maier, A. Zündorf. Yet Another Association Implementation. In Giese et al. (eds.), *Proceedings 2nd International Fujaba Days*. Pp. 67–72. 2004. Available at http://www.fujaba.de/fileadmin/Informatik/Fujaba/Resources/Publications/Fujaba_Days/tr-ri-04-253.pdf.

[Pad93] J. Padberg. Survey of High-Level Replacement Systems. 1993. Technical Report, Technische Universität Berlin. See http://citeseer.ist.psu.edu/padberg93survey.html.

[Ren04] A. Rensink. The GROOVE Simulator: A Tool for State Space Generation. In Pfaltz et al. (eds.), *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*. Lecture Notes in Computer Science 3062, pp. 479–485. Springer, 2004.

[Sch97] A. Schürr. Programmed Graph Replacement Systems. In Rozenberg (ed.), *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. Pp. 479–546. World Scientific, 1997.

[Tae97] G. Taentzer. Parallel High-Level Replacement Systems. *TCS* 186(1-2):43–81, 1997.

[VB07] D. Varró, A. Balogh. The model transformation language of the VIATRA2 framework. *Sci. Comput. Program.* 68(3):214–234, 2007.

[Zün01] A. Zündorf. Rigorous Object Oriented Software Development. 2001. Habilitation Thesis. Universität Paderborn.