Electronic Communications of the EASST
Volume of the Pre-proceedings of GT-VMT (2010)

Preliminary 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 struture 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. We show that the resulting graphs still form an adhesive HLR category, and so the standard results from algebraic graph transformation apply. We show how lists can be manipulated. 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.

* Supported by the EU Artemis project CHARTER

Examples of enriched graph formalisms, introduced exactly for the reason of modelling particular kinds of structures more naturally, are: attributed graphs [EEPT06a], hierarchical graphs [DHP02], and hypergraphs [Hab92]. There is a price to pay for such 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. It follows that enrichments in the graph formalism are only justified if the resulting increase in complexity is outweighed by the corresponding advantages in modelling.

Fortunately, on the theoretical level there is a touchstone against which we can measure the feasibility of any enriched graph formalism: namely, the concept of adhesive HLR categories, developed in [EPPH06] — see [EEPT06b] for an extensive description. That is, if one can prove that an enriched notion of graphs satisfies the constraints of HLR-adhesiveness, then there is a standard way to define their transformation, and many desirable properties are known to hold.

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. Thus, in line with the reasoning above, it makes sense to solve the problem of modelling list structures by enriching the graphs. In order to justify the cost of a more complex formalism on the level of theory, we show that the resulting category of graphs is still adhesive HLR.

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 (HLR adhesiveness of the resulting category). 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 two events (1500 m and 5 km for men). The notation means that the 1500 m event has four participants, in the order from top to bottom, whereas the 5 km event has three, namely Kramer, Tuitert and Davis (in that order). A third event, 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.

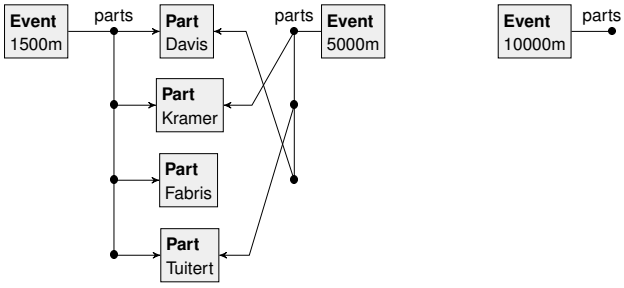


Figure 1: Two skating events with overlapping lists of participants

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 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.

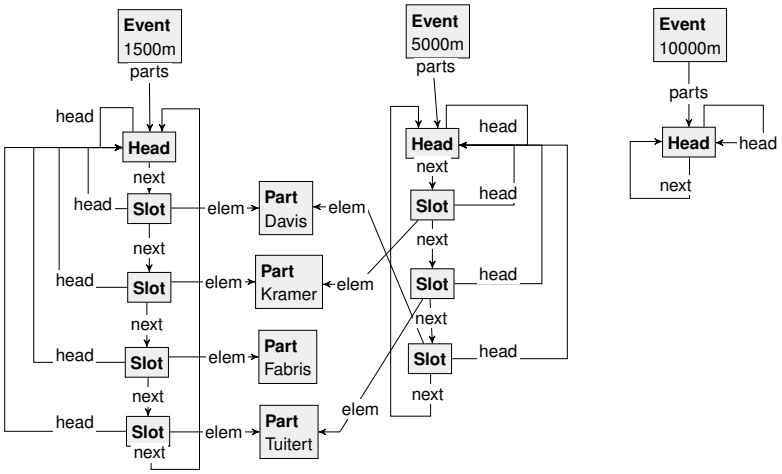


Figure 2: Plain graph representation of the structure in Figure 1

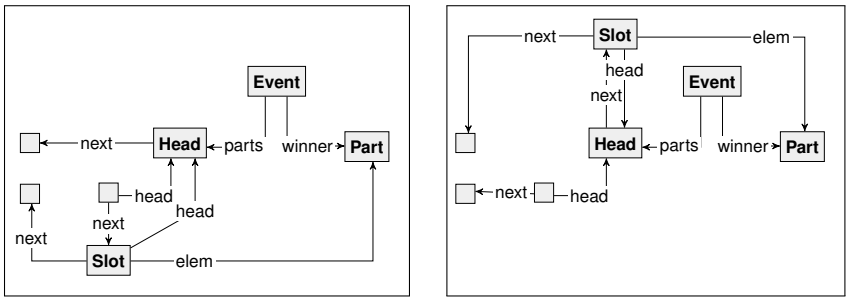


Figure 3: Plain graph rule moving the winner of an event to the top of the list.

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 abstract, declarative way of manipulating structures. To name one consequence, 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.

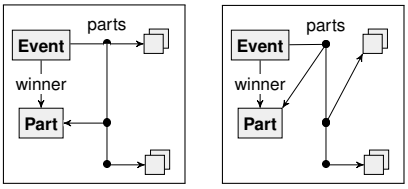


Figure 4: List graph rule moving the winner of an event to the top of the list.

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.

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 can be seen as standing for arbitrary sequences of nodes from the host graph. List nodes can only occur within edge targets, never as sources. Graph morphisms are extended to list nodes as follows: every list node is matched either by a sequence of plain nodes, or by a single list node. This is extended to list edges in the natural way.

For instance, Figure 4 shows the rule performing the same operation as the one in 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 will extend a standard representation of multi-sorted graphs with list nodes and list edges, and we will show that the result is an adhesive HLR category [EPPH06]. We will

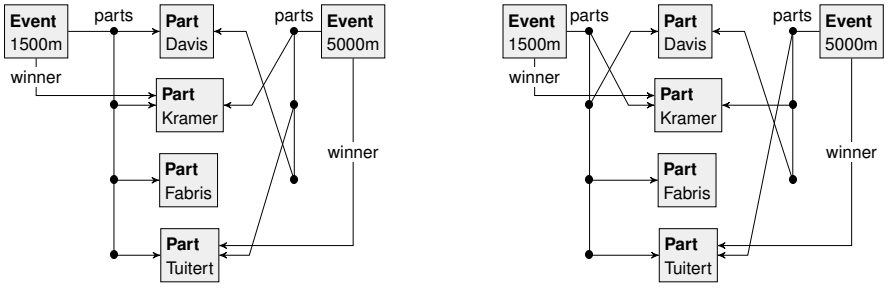


Figure 5: Applying Figure 4 twice to the left hand side graph yields the right hand side graph

use double push-outs (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 to be defined category. A homomorphism $f : G \to H$ is a structure preserving mapping of nodes and edges. In our case, three mappings have to be defined: (1) one for plain nodes, which are mapped to plain nodes; (2) one for list nodes, which are mapped either to list nodes or to sequences of plain nodes; and (3) one for list edges, which 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 (Definition 2) as arrows and composition as in Definition 3. The identity arrows are the homomorphisms that are pairs of identity functions.

The next step is to show that the constructed $\mathbb{GL}$ is also an adhesive HLR category, which allows DPO graph rewriting to be defined in it. First, we briefly recall the definitions:

Definition 5 (*van Kampen squares*)
A pushout p is a VK-square if for any commutative cube where the bottom face is p, it holds that the top face is a pushout iff the front faces are pullbacks.

Definition 6 (*adhesive HLR categories*)
A category $\mathbf{C}$ with a given subclass of M-morphisms is an adhesive HLR-category, iff:
- M is a class of monomorphisms that is closed under isomorphisms, composition and decomposition
- all objects have pullbacks and pushouts along M-morphisms, and M-morphisms are closed under pushouts and pullbacks
- all pushouts form VK-squares

In order to show that $\mathbb{GL}$ is an adhesive HLR category, we have to identify a subclass of M-morphisms and prove the properties that are listed in Definition 6. In this paper, we will present a part of the proof only; the full proof can be found in [MR10].

Definition 7 (*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: an M-morphism does not perform matching of list nodes to sequences, but maps them one-to-one to list nodes only.

Theorem 1 (*$\mathbb{GL}$ is an adhesive HLR category*)
$\mathbb{GL}$ is an adhesive HLR category.

Proof (sketch). In this paper, we limit ourselves to sketching the construction of pullbacks and

pushouts. The rest of the proof can be found in [MR10].

- *Construction of pullbacks.* 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.
- *Construction of pushouts.* 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.

This result allows DPO graph rewriting to be applied in our category $\mathbb{GL}$.

Definition 8 (*double pushout rewriting*)
A graph production $L \xleftarrow{l} K \xrightarrow{r} R$ is applied to a host graph G with the following procedure:

- First, find a morphism m that maps L to G.
- Then, find 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.

Note furthermore that:

- If either of the morphisms m or k does not exist, the rule cannot be applied.
- In an adhesive HLR category, when l and r are both monomorphisms from the subclass M, morphism k is unique (if it exists). This ensures that the result of applying the rule is determined completely (up to isomorphism) by m.

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 1), which copies $(B \backslash A)$ (in this case $(L \backslash K)$ into the pushout (in this case the intermediate host graph D).

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 9 (*surjective M-morphisms*)

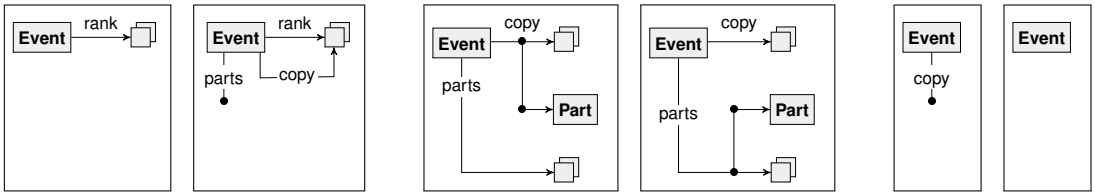


Figure 6: List graph rules creating a reversed parts list out of a rank list.

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 10 (*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 [MR10]. 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 three rules in Figure 6.

- 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 7 shows a sequence of applications of these rules.

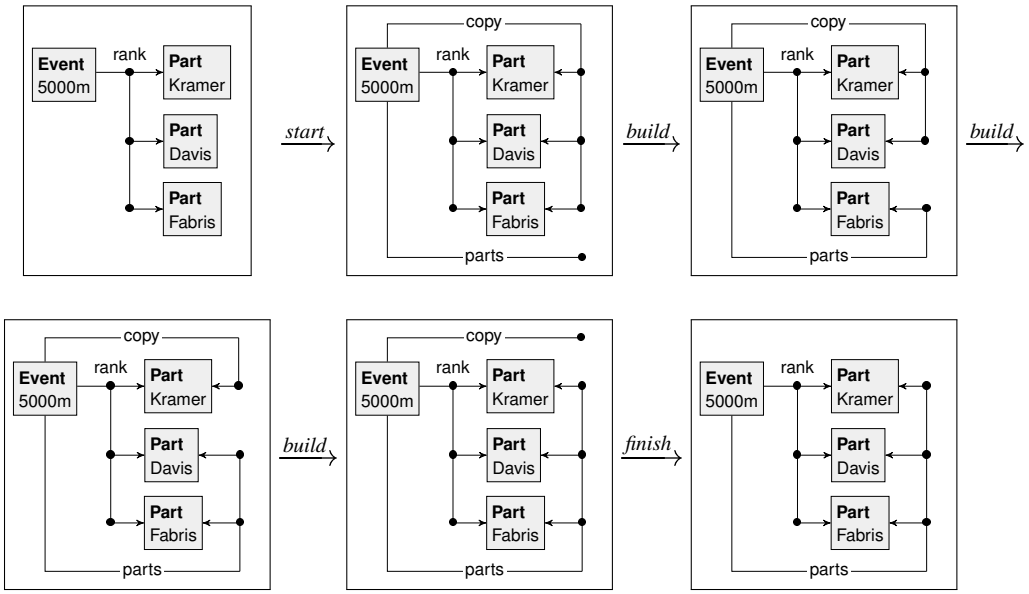


Figure 7: Example production sequence for the rules in Figure 6.

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 also 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 list edges break the usual symmetrical treatment of edge sources and targets, since lists may only occur at the target and not at the source of an edge. 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

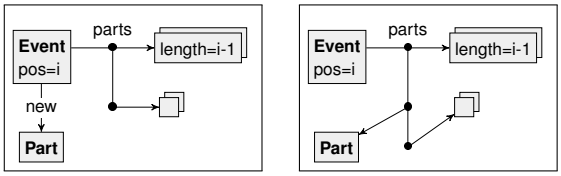


Figure 8: List graph rule inserting an element at a specified position.

extends to standard hyperedges (keeping our specialised notion of morphism), which do not have a distinguished source node at all.

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

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

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 8 specifies that a Part-node should be inserted at index i.

Bibliography

[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.

[EPPH06] H. Ehrig, J. Padberg, U. Prange, A. Habel. Adhesive High-Level Replacement Systems: A New Categorical Framework for Graph Transformation. *Fundam. Inf.* 74(1):1–29, 2006.

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

[MR10] M. de Mol, A. Rensink. 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.

[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.

Electronic Communications of the EASST
Volume of the Pre-proceedings of GT-VMT (2010)

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

Graph Algebras for Bigraphs

Davide Grohmann, Marino Miculan

18 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

Graph Algebras for Bigraphs

Davide Grohmann[1], **Marino Miculan**[2]

[1] grohmann@dimi.uniud.it, [2] miculan@dimi.uniud.it
Department of Mathematics and Computer Science, University of Udine, Italy

Abstract: Binding bigraphs are a graphical formalism intended to be a meta-model for mobile, concurrent and communicating systems. In this paper we present an algebra of *typed graph terms* which correspond precisely to binding bigraphs over a given signature. As particular cases, pure bigraphs and local bigraphs are described by two sublanguages which can be given a simple syntactic characterization.

Moreover, we give a formal connection between these languages and *Synchronized Hyperedge Replacement* algebras and the hierarchical graphs used in *Architectural Design Rewriting*. This allows to transfer results and constructions among formalisms which have been developed independently, e.g., the systematic definition of congruent bisimulations for SHR graphs via the IPO construction.

Keywords: Bigraphs, graph grammars, types.

1 Introduction

Bigraphical Reactive Systems (BRSs) [Mil01] have been proposed as a promising meta-model for ubiquitous, mobile systems. The key feature of BRSs is that their states are *bigraphs*, semi-structured data which can represent at once both the (physical, logical) location and the connections of the components of a system. The dynamics of the system is given by a set of rewrite rules on this semi-structured data.

Bigraphs are very flexible: they have been successfully used for representing many domain-specific calculi and models, from traditional programming languages, to process calculi for concurrency and mobility, from context-aware systems to web-service orchestration languages—see e.g. [JM03, LM06, BDE+06, GM07, BGH+08]. In fact, many variants of bigraphs have been proposed: the original *pure* bigraphs have been later generalized into *binding bigraphs*, allowing also for name scoping; other variants have been proposed, such as *local bigraphs* used for studying the λ-calculus.

In this paper, we propose a *general typed language* for binding bigraphs, which we recall in Section 2. More precisely, in Section 3 we define an algebra of typed graph terms, so that well-typed terms correspond to binding bigraphs, and congruence captures bigraphic equality; this interpretation and corresponding properties are exposed in Section 4. Moreover, as we will show in Section 5, the important subcategories of pure, local and prime bigraphs can be described by suitable sublanguages which can be given a simple and effective syntactic characterization.

Finally, we show how this language for binding bigraphs can be tailored to formalisms introduced in literature (for quite different purposes). In Section 6 we consider hypergraphs used in *Synchronized Hyperedge Rewriting* [FHL+05] and the "designs" of *Architectural Design Rewriting* [BLMT07].

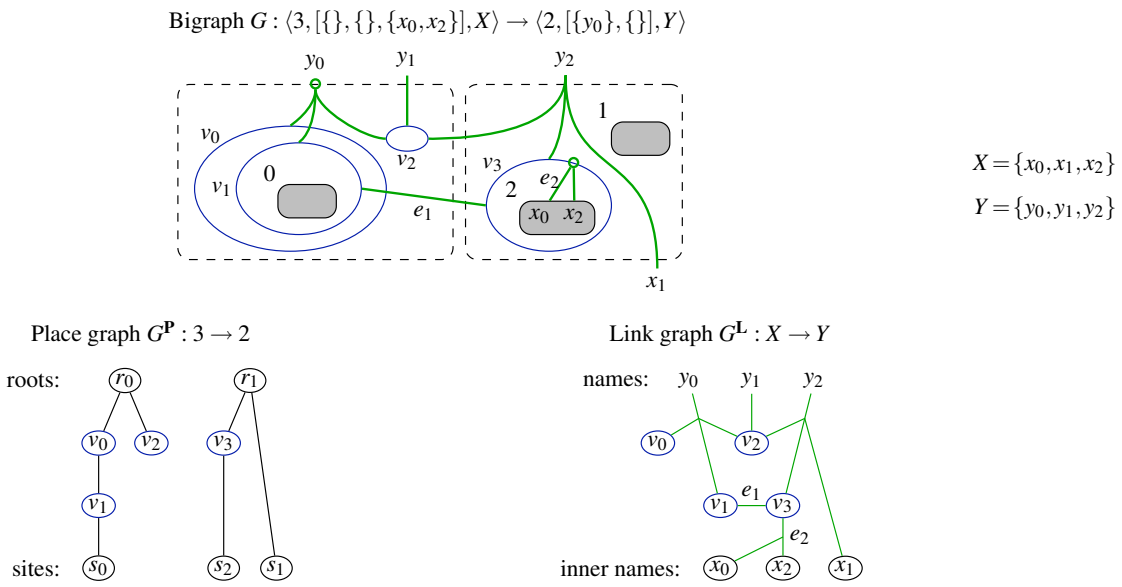


Figure 1: A binding bigraph (picture taken from [BDGM07]).

These results are useful for several reasons. First, the typed algebra we propose can be used as a language for binding, pure, local and prime bigraphs, alternative to the bigraph algebra [JM03]. Moreover, we confirm once more that bigraphs are a quite expressive general framework of ubiquitous systems. These connections pave the way for transferring results and constructions from bigraphs to the SHR and ADR frameworks, and vice versa, as suggested in Section 7.

2 Binding Bigraphs

In this section we recall Milner's *binding bigraphs* [JM03, JM04]. Intuitively, a binding bigraph represents an open system, so it has an inner and an outer interface to "interact" with subsystems and the surrounding environment (Figure 1). The *width* of the outer interface describes the *roots*, that is, the various locations containing the system components; the width of the inner interface describes the *sites*, that is, the holes where other bigraphs can be inserted. On the other hand, the *names* in the interfaces describe free links, that is end points where links from the outside world can be pasted, creating new links among nodes. In particular, we consider binding bigraphs with (possibly) multiply localized names, as in [Mil04] and slightly generalizing [JM03, JM04].

More formally, let $\mathscr{K}$ be a *binding signature* of controls (i.e., node types), and $ar\colon \mathscr{K} \to \mathbb{N} \times \mathbb{N}$ be the arity function. The arity pair (h,k) (written $h \to k$) consists of the *binding arity* h and the *free arity* k, indexing respectively the binding and the free ports of a control.

Definition 1 (Interfaces) An *interface* is a pair $\langle m, X\rangle$ where m is a finite ordinal (called *width*), X is a finite set of names. A *binding interface* is a triple $\langle m, loc, X\rangle$, where $\langle m, X\rangle$ is an interface and $loc \subseteq m \times X$ is a *locality* map associating a subset of the names in X with sites in m. If $(s,x) \in loc$ then x is *located* at s, or *local* to s; x is *global* if, $\forall s$, $(s,x) \notin loc$.

Sometime, we shall represent the locality map as a vector $\vec{X} = (X_0, \ldots, X_{m-1})$ of subsets, where X_s is the set of names local to s; thus $X \setminus \vec{X} = X \setminus (X_0 \cup \cdots \cup X_{m-1})$ are the global names. We call an interface *local* (resp. *global*) if all its names are local (resp. global). We denote by $\uplus$ the union of already disjoint sets, i.e., $S \uplus T \triangleq S \cup T$ if $S \cap T = \emptyset$, otherwise it is undefined.

Definition 2 (Pure and binding bigraphs) A *(pure) bigraph* $G\colon \langle m, X\rangle \to \langle n, Y\rangle$ is composed by a *place graph* G^P and a *link graph* G^L describing node nesting and (hyper-)links among nodes:

$$\begin{aligned} G &= (V, E, ctrl, G^P, G^L)\colon \langle m, X\rangle \to \langle n, Y\rangle && \text{(pure bigraph)} \\ G^P &= (V, ctrl, prnt)\colon m \to n && \text{(place graph)} \\ G^L &= (V, E, ctrl, link)\colon X \to Y && \text{(link graph)} \end{aligned}$$

where V, E are the sets of nodes and edges respectively; $ctrl\colon V \to \mathcal{K}$ is the *control map,* which assigns a control to each node; $prnt\colon m \uplus V \to V \uplus n$ is the (acyclic) *parent map* (often written also as $<$); $link\colon X \uplus P \to E \uplus B \uplus Y$ is the *link map,* where $P = \sum_{v \in V} \pi_1(ar(ctrl(v)))$ is the set of ports and $B = \sum_{v \in V} \pi_2(ar(ctrl(v)))$ is the set of bindings (associated to all nodes). A link $l \in X \uplus P$ is *bound* if $link(l) \in B$; it is *free* if $link(l) \in Y \uplus E$.

A *binding bigraph* $G\colon \langle m, loc, X\rangle \to \langle n, loc', Y\rangle$ is a (pure) bigraph $G^u\colon \langle m, X\rangle \to \langle n, Y\rangle$ satisfying the following locality conditions:

1. if a link is bound, then the names and ports linked to it must lie within the node that binds it;
2. if a link is free, with outer name x, then x must be located in every region that contains any inner name or port of the link.

Definition 3 (Binding bigraph category) The category of binding bigraphs over a signature $\mathcal{K}$ (written $\mathbf{Bbg}(\mathcal{K})$) has local interfaces as objects, and binding bigraphs as morphisms.

Given two binding bigraphs $G\colon \langle m, loc, X\rangle \to \langle n, loc', Y\rangle$, $H\colon \langle n, loc', Y\rangle \to \langle k, loc'', Z\rangle$, the composition $H \circ G\colon \langle m, loc, X\rangle \to \langle k, loc'', Z\rangle$ is defined by composing their place and link graphs, whenever they have disjoint node and edge sets:

1. the composition of $G^P\colon m \to n$ and $H^P\colon n \to k$ is defined as

$$H^P \circ G^P = (V_G \uplus V_H, ctrl_G \uplus ctrl_H, (id_{V_G} \uplus prnt_H) \circ (prnt_G \uplus id_{V_H}))\colon n \to k;$$

2. the composition of $G^L\colon X \to Y$ and $H^L\colon Y \to Z$ is defined as

$$H^L \circ G^L = (V_G \uplus V_H, E_G \uplus E_H, ctrl_G \uplus ctrl_H, (id_{E_G} \uplus link_H) \circ (link_G \uplus id_{P_H}))\colon X \to Z.$$

Definition 4 (Categories of pure, local and prime bigraphs) The category of *pure bigraphs* (**Big**) is the full subcategory of binding bigraphs whose objects are of the form $\langle n, (\emptyset), X\rangle$ (often shorten as $\langle n, X\rangle$).

The category of *local bigraphs* (**Lbg**) is the full subcategory of binding bigraphs whose objects are of the form $\langle n, (\vec{X}), \bigcup \vec{X}\rangle$ (often shorten as $(\vec{X})$).

The category of *prime bigraphs* (**Pbg**) is the full subcategory of local bigraphs whose objects are of the form $\langle n, (\vec{X}), \bigcup \vec{X}\rangle$, with $n \in \{0, 1\}$, (often shorten as before: $(\vec{X})$).

Intuitively, in pure bigraphs all names are global, whilst in local bigraphs all names are local, finally prime bigraphs are all the local bigraphs with one root, and one or zero holes.

An important operation about (bi)graphs, is the *tensor product*. Intuitively, the tensor product puts "side by side" two bigraphs, given $G\colon \langle m,(\vec{X}),X\rangle \to \langle n,(\vec{Y}),Y\rangle$ and $H\colon \langle m',(\vec{X}'),X'\rangle \to \langle n',(\vec{Y}'),Y'\rangle$, their tensor product is a bigraph $G\otimes H\colon \langle m+m',(\vec{X}\vec{X}'),X\cup X'\rangle \to \langle n+n',(\vec{Y}\vec{Y}'),Y\cup Y'\rangle$ defined when global names in X,X' and Y,Y' are disjoint. Two useful variants of tensor product can be defined using tensor and composition: the *parallel product* $||$, which merges shared names between two bigraphs, and the *prime product* $|$, that also merges all roots in a single one. As shown in [Mil04, DB06], all bigraphs can be constructed by composition and tensor product from a set of *elementary bigraphs:*

- $1 : \langle 0,(\emptyset),\emptyset\rangle \to \langle 1,(\emptyset),\emptyset\rangle$ is the barren (i.e., empty) root;
- $merge_n : \langle n,(\emptyset),\emptyset\rangle \to \langle 1,(\emptyset),\emptyset\rangle$ merges n roots into a single one;
- $\gamma_{m,n,(\vec{X},\vec{Y})} : \langle m+n,(\vec{X}\vec{Y}),(\bigcup\vec{X})\cup(\bigcup\vec{Y})\rangle \to \langle m+n,(\vec{Y}\vec{X}),(\bigcup\vec{X})\cup(\bigcup\vec{Y})\rangle$ permutes the first m roots having local names in $\vec{X}$ with the following n roots with local names in $\vec{Y}$.
- $/x : \langle 0,(\emptyset),\{x\}\rangle \to \langle 0,(\emptyset),\emptyset\rangle$ is a *closure*, that is it maps x to an edge;
- $y/X : \langle 0,(\emptyset),X\rangle \to \langle 0,(\emptyset),\{y\}\rangle$ substitutes the names in X with y, i.e., it maps the whole set X to y; as a shortcuts, we write $\vec{y}/\vec{X}$ to mean $y_0/X_0\otimes\ldots\otimes y_{n-1}/X_{n-1}$;
- $\ulcorner X\urcorner : \langle 1,(X),X\rangle \to \langle 1,(\emptyset),X\rangle$ means that names in X are switched from local to global
- conversely, $(X) : \langle 1,(\emptyset),X\rangle \to \langle 1,(X),X\rangle$ localizes the global names of X.
- Finally, $K_{\vec{x}(\vec{X})} : \langle 1,(X),\emptyset\rangle \to \langle 1,(\emptyset),\vec{x}\rangle$ is a control which may contain other graphs, and it has free ports linked to the name in $\vec{x}$, whilst the names $\vec{X}$ are connected to its binding ports.

We use the convention that local names are enclosed in parenthesises.

Bigraphs can be given always in *discrete normal form*: the idea of this normal form is to separate wirings (i.e., linkings) from discrete bigraphs (i.e., nesting of nodes). The following is an easy generalization of [DB06, Theorem 1] to the case of bigraphs with multiply located names.

Theorem 1 (Discrete Normal Form (DNF)) *1. Any binding bigraph* $G : I \to \langle n,\vec{Y}_B,(\bigcup\vec{Y}_B)\uplus Y_F\rangle$ *can be expressed as*

$$G = \left(\bigotimes_{i<n}(\vec{y}_i)/(\vec{X}_i)\otimes\bigotimes_{i<|Y_F|}w_i/W_i\otimes\bigotimes_{i<|Z|}(/z_i\circ z_i/Z_i)\right)\circ D$$

where $D : I \to \langle m,\vec{X},(\bigcup\vec{X})\uplus W\uplus Z\rangle$ *is a name discrete.*

2. Any name discrete $D : I \to \langle m,\vec{X},(\bigcup\vec{X})\uplus W\uplus Z\rangle$ *can be expressed as*

$$D = \alpha\otimes((P_0\otimes\ldots\otimes P_{n-1})\circ\pi)$$

where α *is a renaming, and* π *a permutation.*

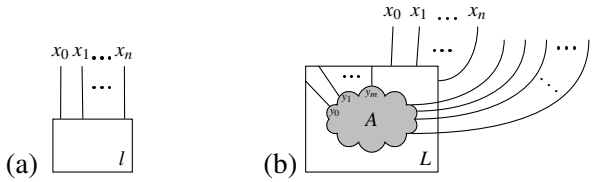


Figure 2: Example of an atomic label (a) and a non-atomic one (b).

3. *Any name-discrete prime* $P : J \to \langle 1, (U_B), U \rangle$ *can be expressed as*

$$(U_B) \circ (merge_{n+k} \otimes id_U) \circ (\ulcorner \alpha_0 \urcorner \otimes \ldots \otimes \ulcorner \alpha_{n-1} \urcorner \otimes M_0 \otimes \ldots \otimes M_{k-1}) \circ \pi$$

where every $M_i : J_i \to \langle 1, (\emptyset), U_i^M \rangle$ *is a free discrete molecule, and for renamings* $\alpha_i : V_i \to U_i^C$, *we have* $U = (\biguplus_{i \in n} U_i^C) \uplus \biguplus_{i \in k} U_i^M$.

4. *Any free discrete molecule* $M : K \to \langle 1, (\emptyset), \vec{x} \uplus V \rangle$ *can be expressed as*

$$M = (K_{\vec{x}(\vec{S})} \otimes id_V) \circ P$$

where $P : H \to \langle 1, (\biguplus \vec{S}), (\biguplus \vec{S}) \uplus V$ *is a name discrete.*

3 Graph Grammar for Bigraphs

In this section we introduce a language for binding bigraphs. It is parametric over a ranked alphabet of labels $\mathscr{L} = (\mathscr{L}_a, \mathscr{L}_n, exit : \mathscr{L}_a \cup \mathscr{L}_n \to \mathbb{N}, in : \mathscr{L}_n \to \mathbb{N})$, where $\mathscr{L}_a$ are the *atomic* labels, ranged over by l, and $\mathscr{L}_n$ are the *non-atomic* labels, ranged over by L. Each label is given an *exit-rank*, $exit(l) \in \mathbb{N}$, enumerating the "exiting tentacles". Non-atomic labels have an *in-rank* $in(L) \in \mathbb{N}$, enumerating the label's "incoming tentacles". We often denote by $\mathscr{L}$ the set $\mathscr{L}_a \cup \mathscr{L}_n$.

One may think of a node with an atomic label l as an hyperedge with $exit(l)$ tentacles, as in Figure 2(a). A node labelled with L has $exit(L)$ tentacles, and may contain a subsystem whose exiting tentacles are either linked to the $in(L)$ ports of the node, or go "outside" the node, see Figure 2(b). More formally, the language of graphs is as follows.

Definition 5 (Agent graphs) Let $\mathscr{N}$ be an infinite set of names, $\mathscr{V}$ be an infinite set of variables, and $\mathscr{L}$ be a ranked alphabet of labels. An *agent-graph* A is a term generated as follows:

$$A ::= \varepsilon \mid \mathbf{0} \mid l(\vec{x}) \mid L(\vec{x})[A \backslash \vec{y}] \mid X \mid A|A \mid A \parallel A \mid \nu z.A \mid A[w \mapsto z] \mid A \upharpoondown z \mid A \upharpoonup z$$

where $\vec{x}, \vec{y} \subseteq \mathscr{N}^*$; $l \in \mathscr{L}_a$, $L \in \mathscr{L}_n$ with $exit(l) = exit(L) = |\vec{x}|$, $in(L) = |\vec{y}|$; $X \in \mathscr{V}$; and $w, z \in \mathscr{N}$. Moreover, in a term A, each X is used at most once.

Intuitively, ε represents the absence of any system, that is, no agents at all, while $\mathbf{0}$ represents an empty agent (i.e., an agent with no nodes). We denote by $l(\vec{x})$ atomic hyperedges whose tentacles are linked to the names in $\vec{x}$, whilst by $L(\vec{x})[A \backslash \vec{y}]$ non-atomic hyperedges having exiting tentacles linked to the names in $\vec{x}$, containing a subgraph A whose names $\vec{y}$ are linked on the edge itself. Graph variables X are needed for representing open systems, i.e., graphs with holes.

$$\frac{}{\langle\rangle;\tau\vdash\varepsilon:(\langle\rangle,\tau)}\qquad\frac{}{\langle\rangle;\tau\vdash\mathbf{0}:(\emptyset,\tau)}\qquad\frac{}{\langle\rangle;\tau\vdash l(\vec{x}):(\emptyset,\vec{x}\cup\tau)}\qquad\frac{\Gamma;\tau\vdash A:(\sigma\cup\vec{y},\rho)\quad\vec{y}\cap\sigma=\emptyset}{\Gamma;\tau\vdash L(\vec{x})[A\backslash\vec{y}]:(\sigma,\rho\cup\vec{x})}$$

$$\frac{\Gamma;\tau\vdash A:(\sigma,\rho)\;\Gamma';\tau'\vdash A':(\sigma',\rho')\;\tau\cap\tau'=\emptyset}{\Gamma,\Gamma';\tau\cup\tau'\vdash A\mid A':(\sigma\cup\sigma',\rho\cup\rho')}\qquad\frac{\Gamma;\tau\vdash A:(\vec{\sigma},\rho)\;\Gamma';\tau'\vdash A':(\vec{\sigma}',\rho')\;\tau\cap\tau'=\emptyset}{\Gamma,\Gamma';\tau\cup\tau'\vdash A\mid\mid A':(\vec{\sigma}\vec{\sigma}',\rho\cup\rho')}$$

$$\frac{}{X:\sigma;\tau\vdash X:(\sigma,\tau)}\qquad\frac{\Gamma;\tau\vdash A:(\vec{\sigma},\rho)}{\Gamma;\tau\vdash\nu x.A:(\vec{\sigma}\setminus\{x\},\rho\setminus\{x\})}\qquad\frac{\Gamma;\tau\vdash A:(\vec{\sigma},\rho)\quad\pi\text{ permutation}}{\pi(\Gamma);\tau\vdash A:(\vec{\sigma},\rho)}$$

$$\frac{\Gamma;\tau\vdash A:(\vec{\sigma},\rho)\quad x\in\vec{\sigma}}{\Gamma;\tau\vdash A[x\mapsto y]:(\vec{\sigma}\{y/x\},\rho)}\qquad\frac{\Gamma;\tau\vdash A:(\vec{\sigma},\rho)\quad x\notin\vec{\sigma}}{\Gamma;\tau\vdash A[x\mapsto y]:(\vec{\sigma},(\rho\setminus\{x\})\cup\{y\})}$$

$$\frac{\Gamma;\tau\vdash A:(\sigma,\rho\cup\{x\})\quad x\notin\rho}{\Gamma;\tau\vdash A\downarrow x:(\sigma\cup\{x\},\rho)}\qquad\frac{\Gamma;\tau\vdash A:(\vec{\sigma},\rho)\quad x\in\vec{\sigma}}{\Gamma;\tau\vdash A\uparrow x:(\vec{\sigma}\setminus\{x\},\rho\cup\{x\})}$$

Figure 3: Type inference rules for agent-graphs.

Two agent graphs A,B can be composed in parallel in two different ways: $A\mid B$ "merges" two graphs into a unique one (i.e., in the same location), while $A\mid\mid B$ puts the two systems side by side, i.e., they keep living in different locations.

As usual, $\nu y.A$ limits the scope of y to A, while $A[w\mapsto z]$ is the explicit substitution of name w with z. Notice that the agent-graph $A[w\mapsto z]$ exhibits the name z also when w does not appear in A; in this case, the operator $\_[w\mapsto z]$ "creates" unused (or idle) names.

Finally, $A\downarrow z$ *localizes* z to (the location of) A. This means that z can only be accessed by/linked to nodes lying in the location of A, that is, they must be inside or in parallel ($\mid$) to A. Dually, $A\uparrow z$ *globalizes* z, allowing a localized name to be used also by nodes in different locations.

From the above intuition, it is clear that not all terms generated by this grammar are meaningful. For instance, what is the meaning of $A|B$ or $A\downarrow z$ when A is a system with more than one location? In order to rule out meaningless terms, we introduce a typing system for agent graphs.

Definition 6 (Type system for agent graphs) *Simple types* τ,σ,ρ are finite sets of names.

An *agent-type* $(\vec{\tau},\tau)$ is a pair formed by a list $\vec{\tau}=\tau_0\ldots\tau_{n-1}$ of simple types (where $\langle\rangle$ is the empty list), and a simple-type τ, such that $\tau\cap(\tau_0\cup\ldots\cup\tau_{n-1})=\emptyset$.

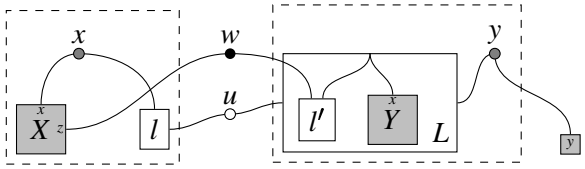
An *environment* is a pair $\Gamma;\tau$ formed by a list of typed variables ($\Gamma=\vec{X}:\vec{\tau}=X_0:\tau_0,\ldots,X_{n-1}:\tau_{n-1}$) and a simple-type ($\tau$), such that $\tau\cap(\tau_0\cup\ldots\cup\tau_{n-1})=\emptyset$.

A typing judgement is of the form $\Gamma;\tau\vdash A:(\vec{\sigma},\rho)$, whose inference rules are in Figure 3.

Agent-types $(\vec{\tau},\tau)=(\tau_0\ldots\tau_{n-1},\tau)$ describe both the locations of a graph, and the names that the graph exposes to the environment. Names in τ are "global", and can be used from every location; instead, names in τ_i can be used only inside the i-th location of the system.

We are interested in open systems, that is systems with "holes". An environment $\Gamma;\tau=X_0:\tau_0,\ldots,X_{n-1}:\tau_{n-1};\tau$ declares the "inner interface" of an agent: the names of the variables (X_i for $i\in n$), with their sets of local names (τ_i are the names local to X_i), and the set of incoming global names (τ), i.e., names that can be used from within any variable.

Notation. List concatenation is denoted simply by juxtaposition. We extend the operators $\in$,



$$Y : \{x\}, X : \{x,z\}; \{y\} \vdash \nu u.((l(x,u) \mid X) \downarrow z[z \mapsto w] \parallel L(y,u)[(Y \mid l'(w,x))\backslash x] \downarrow w) : (\{w\}\{w\}, \{x,y\})$$

Figure 4: An example of an agent-graph.

$\cup$ and $\setminus$ over set lists as follows: $x \in \vec{\tau}$ iff there exists $\bar{\tau} \in \vec{\tau}$ such that $x \in \bar{\tau}$; let S be a set, then $\tau_0 \ldots \tau_{n-1} \cup S \triangleq (\tau_0 \cup S) \ldots (\tau_{n-1} \cup S)$ and $\tau_0 \ldots \tau_{n-1} \setminus S \triangleq (\tau_0 \setminus S) \ldots (\tau_{n-1} \setminus S)$. Γ_1, Γ_2 is the concatenation of Γ_1 and Γ_2, defined when $\mathrm{dom}(\Gamma_1) \cap \mathrm{dom}(\Gamma_2) = \emptyset$. We introduce some useful shortcuts: $\nu\vec{x}.A = \nu x_{|\vec{x}-1|} \ldots \nu x_0.A$; $A[\vec{x} \mapsto \vec{y}] = A[x_0 \mapsto y_0] \ldots [x_{n-1} \mapsto y_{n-1}]$ when $|\vec{x}| = |\vec{y}| = n$; $A[X \mapsto x] = A[x_0 \mapsto x] \ldots [x_{|X|} \mapsto x]$ if $X \neq \emptyset$, $A[\emptyset \mapsto x] = A[z \mapsto x]$ for some z fresh (i.e., z is not used by A); finally $A[\vec{X} \mapsto \vec{x}] = A[X_0 \mapsto x_0] \ldots [X_{n-1} \mapsto x_{n-1}]$ when $|\vec{X}| = |\vec{x}| = n$.

Some intuitive explanation of the typing rules may be useful. Empty agents have only global names, as defined by the environment. Notice that $\mathbf{0}$ is the null process, which *is* an agent, while ε is no agent at all. An atomic hyperedge whose exit-tentacles are linked to the names $\vec{x}$ exposes those (global) names to a context, plus the ones added by the environment. As before, a non-atomic hyperedge shows names $\vec{x}$ that are linked to its exit-tentacles, plus the global ones defined in the environment. The difference is that it contains a graph term having $\vec{y}$ local names, that are linked to the hyperedge's input tentacles, and hence they are not visible from the context.

The names exposed by a composition ($|$) of two subgraphs are the union of the names exposed by the two subgraphs. The rule for $\parallel$ is quite similar, but in this case the two graphs keep their different locations, and hence the names can be treated in a different way, so global names are the union of agents' global names, whilst local names remains unchanged, i.e, the two lists of local names are concatenated. If a variable has type σ in an environment Γ, then it exposes σ local names and the global names τ defined by the environment. The restriction deletes a name from the set of global or local exposed names. The next rule describes the possibility to reorder the variables in the environment; it will be important in the definition of a category for agent-graphs.

For the substitution $A[w \mapsto z]$ there are two cases: (1) if w is localized, it will be substituted by z; (2) if it is global the substitution (possibly) deletes w and adds z to the set of global names. Notice that if w,z are not used in A, then it effectively adds the name z to its interface.

An example of an agent-graph is given in Figure 4, where white nodes are closed (that is, nodes not accessible from the context); the other are the external nodes (which can be visible by a context): the grey nodes are global and the black ones are local.

Now, we can prove the following properties on our language.

Proposition 1 *If* $\Gamma; \tau \vdash A : (\vec{\sigma}, \tau)$ *and* $\Gamma; \tau \vdash A : (\vec{\sigma}', \tau')$ *then* $\vec{\sigma} = \vec{\sigma}'$ *and* $\tau = \tau'$.

Lemma 1 (substitution lemma) *The following rule is admissible.*

$$\frac{\Gamma_i; \tau_i \vdash A_i : (\sigma_i, \rho_i) \quad 0 \leq i < n \quad \forall i \neq j.\, \tau_i \cap \tau_j = \emptyset \quad X_0 : \sigma_0, \ldots, X_{n-1} : \sigma_{n-1}; \bigcup_{i=0}^{n-1} \rho_i \vdash A : (\vec{\eta}, \zeta)}{\Gamma_0, \ldots, \Gamma_{n-1}; \bigcup_{i=0}^{n-1} \tau_i \vdash A\{A_0/X_0, \ldots, A_{n-1}/X_{n-1}\} : \vec{\eta}; \zeta}$$

As happens often with graph grammars, the same system may be denoted by many terms. Therefore, it is convenient to introduce a *structural congruence* over terms, capturing graph isomorphisms up-to free nodes. Congruence judgments are of the form $\Gamma;\tau \vdash A \equiv B$, for A,B terms of the language. This turns our language into a *graph algebra*, whose axioms are in Appendix A.

Proposition 2 *Let* $\Gamma;\tau \vdash A \equiv A'$, $\Gamma;\tau \vdash A : (\vec{\sigma},\rho)$ *if and only if* $\Gamma;\tau \vdash A' : (\vec{\sigma},\rho)$.

4 Interpreting Agent Graphs as Binding Bigraphs

In this section we give an interpretation of agent graphs as binding bigraphs, showing that the language is sound and complete, and that the axiomatization captures bigraphical equivalence. In order to simplify the translation from our algebra to bigraphs and back, we first introduce a category for agent-graphs.

Definition 7 The category $\mathbf{A}(\mathscr{L})$ of agent-graphs, over a ranked alphabet $\mathscr{L}$, has graph types $(\vec{\sigma},\rho)$ as objects, and judgments on agent-graphs as morphisms, that is, if $X_0 : \eta_0,\ldots,X_{n-1} : \eta_{n-1};\tau \vdash A : (\vec{\sigma},\rho)$ then $(X_0,\ldots,X_{n-1},A) : (\vec{\eta},\tau) \to (\vec{\sigma},\rho)$ is a morphism. Composition is defined in virtue of Lemma 1.

Proposition 3 $(\mathbf{A}(\mathscr{L}),\|,\langle\rangle;\emptyset \vdash \varepsilon : (\langle\rangle,\emptyset))$ *is a strict symmetric monoidal category.*

4.1 Interpretation of agent-graphs as binding bigraphs

Let $\mathscr{L}$ be a ranked alphabet of labels; we define an interpretation functor from the agent-graph category $\mathbf{A}(\mathscr{L})$ to the binding bigraph category $\mathbf{BBg}(\mathscr{K}_{\mathscr{L}})$, for a suitable bigraphical signature $\mathscr{K}_{\mathscr{L}}$.

The idea is to translate the agent-graph hyperedges into nodes, and nodes (or names) into links (i.e., outer names and edges); hence, the bigraphical signature corresponds to the alphabet of labels. Formally:

$$\mathscr{K}_{\mathscr{L}} \triangleq \{l : 0 \to exit(l) \mid l \in \mathscr{L}_a\} \cup \{L : in(L) \to exit(L) \mid L \in \mathscr{L}_n\}.$$

We can now define the functor $[\![-]\!] : \mathbf{A}(\mathscr{L}) \to \mathbf{BBg}(\mathscr{K}_{\mathscr{L}})$ by induction on the typing judgments:

Objects:
$$[\![(\vec{\sigma},\rho)]\!] = \langle|\vec{\sigma}|,\vec{\sigma},\tau\rangle$$

Morphisms:
$$[\![\langle\rangle;\tau \vdash \varepsilon : (\langle\rangle,\tau)]\!] = id_\tau$$
$$[\![\langle\rangle;\tau \vdash \mathbf{0} : (\emptyset,\tau)]\!] = 1 \parallel id_\tau$$
$$[\![\langle\rangle;\tau \vdash l(\vec{x}) : (\emptyset,\vec{x}\cup\tau)]\!] = l_{\vec{x}} \parallel id_\tau$$
$$[\![\Gamma;\tau \vdash L(\vec{x})[A\backslash\vec{y}] : (\sigma,\rho\cup\vec{x})]\!] = L_{\vec{x},(\vec{y})} \circ [\![\Gamma;\tau \vdash A : (\sigma\cup\vec{y},\rho)]\!]$$
$$[\![X : \sigma;\tau \vdash X : (\sigma,\tau)]\!] = id_{(\sigma)} \parallel id_\tau$$
$$[\![\Gamma,\Gamma';\tau\cup\tau' \vdash A \mid A' : (\sigma\cup\sigma',\rho\cup\rho')]\!] = [\![\Gamma;\tau \vdash A : (\sigma,\rho)]\!] \mid [\![\Gamma';\tau' \vdash A' : (\sigma',\rho')]\!]$$
$$[\![\Gamma,\Gamma';\tau\cup\tau' \vdash A \parallel A' : (\vec{\sigma}\vec{\sigma}',\rho\cup\rho')]\!] = [\![\Gamma;\tau \vdash A : (\vec{\sigma},\rho)]\!] \parallel [\![\Gamma';\tau' \vdash A' : (\vec{\sigma}',\rho')]\!]$$
$$[\![\Gamma;\tau \vdash \nu x.A : (\vec{\sigma}\setminus\{x\},\rho)]\!] = /(x) \circ [\![\Gamma;\tau \vdash A : (\vec{\sigma},\rho)]\!] \qquad (\text{if } x \in \vec{\sigma})$$

$$Y : \{x\}, X : \{x,z\}; \{y\} \vdash \nu u.((l(x,u) \mid X) \downarrow z[z \mapsto w] \parallel L(y,u)[(Y \mid l'(w,x))\backslash x] \downarrow w) : (\{w\}\{w\}, \{x,y\})$$

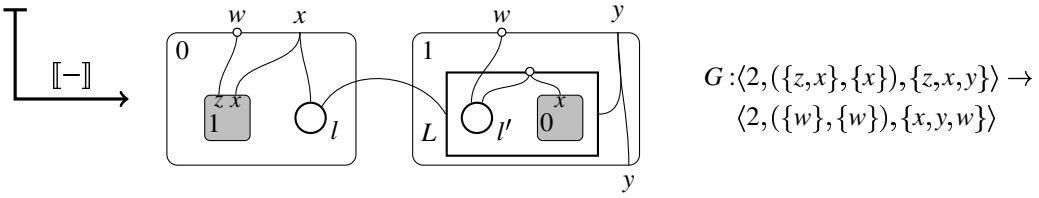


Figure 5: An example of encoding an agent-graph into a binding bigraph.

$$\begin{aligned}
\llbracket \Gamma; \tau \vdash \nu x.A : (\vec{\sigma}, \rho \setminus \{x\}) \rrbracket &= /x \circ (\llbracket \Gamma; \tau \vdash A : (\vec{\sigma}, \rho) \rrbracket \parallel \{x\}) \qquad (\text{if } x \notin \vec{\sigma}) \\
\llbracket \Gamma; \tau \vdash A[x \mapsto y] : (\vec{\sigma}\{y/x\}, \rho) \rrbracket &= (y)/(x) \circ \llbracket \Gamma; \tau \vdash A : (\vec{\sigma}, \rho) \rrbracket \qquad (\text{if } x \in \vec{\sigma}) \\
\llbracket \Gamma; \tau \vdash A[x \mapsto y] : (\vec{\sigma}, (\rho \setminus \{x\}) \cup \{y\}) \rrbracket &= y/x \circ (\llbracket \Gamma; \tau \vdash A : (\vec{\sigma}, \rho) \rrbracket \parallel \{x\}) \qquad (\text{if } x \notin \vec{\sigma}) \\
\llbracket \Gamma; \tau \vdash A \downarrow x : (\sigma \cup \{x\}, \rho) \rrbracket &= (x) \circ \llbracket \Gamma; \tau \vdash A : (\sigma, \rho \cup \{x\}) \rrbracket \\
\llbracket \Gamma; \tau \vdash A \upharpoonright x : (\vec{\sigma} \setminus \{x\}, \rho \cup \{x\}) \rrbracket &= \ulcorner x \urcorner \circ \llbracket \Gamma; \tau \vdash A : (\vec{\sigma}, \rho) \rrbracket \qquad (\text{if } x \in \vec{\sigma}) \\
\llbracket \pi(\Gamma); \tau \vdash A : (\vec{\sigma}, \rho) \rrbracket &= \llbracket \Gamma; \tau \vdash A : (\vec{\sigma}, \rho) \rrbracket \circ \pi .
\end{aligned}$$

Basically, each variable of type σ is encoded as a site having σ local names; therefore, variable permutation is site permutation. Restricted names are represented by bigraph edges, not accessible from the context. The graph $\mathbf{0}$ is represented by the empty root 1. An example is in Figure 5.

We can now prove the following results about this semantics for $\mathscr{L}$. First, it respects the structure of the two categories:

Proposition 4 $\llbracket - \rrbracket : (\mathbf{A}(\mathscr{L}), \|, \langle\rangle; \emptyset \vdash \varepsilon : (\langle\rangle, \emptyset)) \to (\mathbf{Bbg}(\mathscr{K}), \|, id_{(0,\emptyset,\emptyset)})$ *is a strict monoidal functor.*

Moreover, the axiomatization of the graph algebra given in Appendix A is sound and complete with respect to bigraph equivalence.

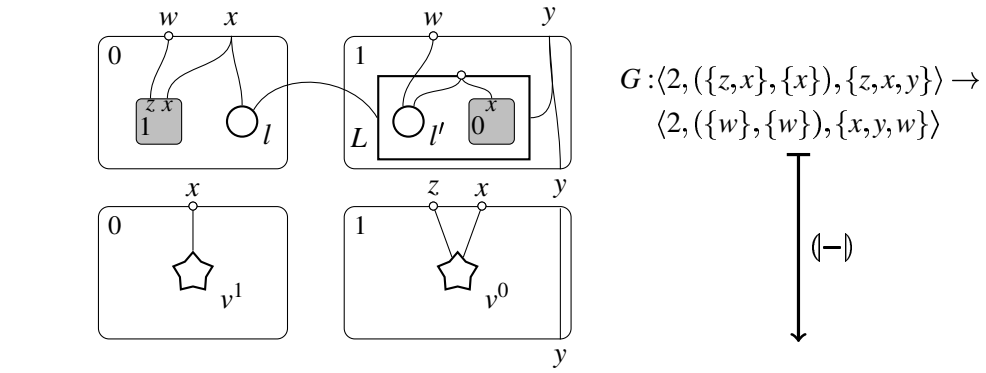
Proposition 5 *Let A, A' be two agent-graphs; then, for every environment $\Gamma; \tau$: $\Gamma; \tau \vdash A \equiv A'$ if and only if $\llbracket \Gamma; \tau \vdash A : (\vec{\sigma}, \rho) \rrbracket = \llbracket \Gamma; \tau \vdash A' : (\vec{\sigma}, \rho) \rrbracket$.*

4.2 Representing binding bigraphs with agent-graphs

In this section we show that our language is expressive enough to cover all binding bigraphs, over a given signature $\mathscr{K}$. To this end, we define a translation from binding bigraphs to agent-graphs of a language whose ranked labels are defined by means of the bigraphical signature.

$$\mathscr{L}_{\mathscr{K}} \triangleq (\{k \mid k : 0 \to n \in \mathscr{K}, atomic\}, \{k \mid k : m \to n \in \mathscr{K}, non\ atomic\}, exit, in)$$
$$exit(k) \triangleq n \quad \text{for } k : m \to n \in \mathscr{K} \qquad in(k) \triangleq m \quad \text{for } k : m \to n \in \mathscr{K} \text{ non atomic}$$

The representation function $(\!|-|\!)$ maps objects of the category $\mathbf{Bbg}(\mathscr{K})$ to agent-types, as $(\!|\langle n, (\vec{X}), (\bigcup \vec{X}) \uplus X \rangle|\!) \triangleq (\vec{X}, X)$. In order to simplify the translation of bigraphs, in virtue of Theorem 1 we can suppose w.l.o.g. that all binding bigraphs are in discrete normal form. Let



$Q_0 : \{x\}, Q_1 : \{x,z\}; \{y\} \vdash$
$\nu z.((((l(x,z) \mid Q_1) \parallel \varepsilon) \downarrow z \parallel (L(y,z)[(Q_0 \mid l'(w,x))[x \mapsto s]\backslash s] \parallel \varepsilon) \downarrow w)[z \mapsto w]) : (\{w\}\{w\}, \{x,y\})$

Figure 6: An example of encoding a binding bigraph into an agent-graph.

$G : \langle m, \vec{X}_B, (\bigcup \vec{X}_B) \uplus X_F \rangle \to \langle n, \vec{Y}_B, (\bigcup \vec{Y}_B) \uplus Y_F \rangle$, be in discrete normal form as follows

$$G = \left(\bigotimes_{i<n} (\vec{y}_i)/(\vec{X}_i) \otimes \bigotimes_{i<|Y_F|} w_i/W_i \otimes \bigotimes_{i<|Z|} (/z_i \circ z_i/Z_i) \right) \circ (\vec{a}/\vec{b} \otimes ((P_0 \otimes \ldots \otimes P_{n-1}) \circ \pi))$$

then, for $\vec{Q} = Q_0, \ldots, Q_{m-1}$ a list of m variables, we define

$$\begin{aligned} (\!|G|\!)_{\vec{Q}} = \nu z_{|Z|-1} \ldots . \nu z_0 . \big(& ((\!|p_0|\!)_{\vec{Q}} \parallel \ldots \parallel (\!|p_{n-1}|\!)_{\vec{Q}}) \\ & [\vec{b} \mapsto \vec{a}][W_0 \mapsto w_0] \ldots [W_{|Y_F|-1} \mapsto w_{|Y_F|-1}][\vec{X}_0 \mapsto \vec{y}_0] \ldots [\vec{X}_{n-1} \mapsto \vec{y}_{n-1}] \big) \end{aligned}$$

where $p_0 \otimes \ldots \otimes p_{n-1} = (P_0 \otimes \ldots \otimes P_{n-1}) \circ \pi \circ (v^0_{(X_0)} \otimes \ldots \otimes v^{m-1}_{(X_{m-1})})$.

Given $p = (U_B) \circ (merge_{h+k} \otimes id_U) \circ (\ulcorner \vec{a}_0/\vec{b}_0 \urcorner \otimes \ldots \otimes \ulcorner \vec{a}_{h-1}/\vec{b}_{h-1} \urcorner \otimes m_0 \otimes \ldots \otimes m_{k-1})$, then

$$\begin{aligned} (\!|p|\!)_{\vec{Q}} = \big(& (\!|K^0_{\vec{x}^0,(\vec{S}^0)} \circ p^0|\!)_{\vec{Q}} \mid \ldots \mid (\!|K^{k_{i-1}}_{\vec{x}^{k_{i-1}},(\vec{S}^{k_{i-1}})} \circ p^{k_{i-1}}|\!)_{\vec{Q}} \mid \\ & (\!|v^{j_0}_{(X_{j_0})}|\!)_{\vec{Q}}[\vec{b}_0 \mapsto \vec{a}_0] \upharpoonright \vec{a}_0 \mid \ldots \mid (\!|v^{j_{h_i-1}}_{(X_{j_{h_i-1}})}|\!)_{\vec{Q}}[\vec{b}_{h_{i-1}} \mapsto \vec{a}_{h_{i-1}}] \upharpoonright \vec{a}_{h_{i-1}} \big) \downarrow U_B \end{aligned}$$

$$(\!|v^i_{(X_i)}|\!)_{\vec{Q}} = Q_i$$

$$(\!|K_{\vec{x},(\emptyset)} \circ 1|\!)_{\vec{Q}} = K(\vec{x}) \qquad \text{where} \quad K \text{ atomic}$$

$$(\!|K_{\vec{x},(\vec{S})} \circ p|\!)_{\vec{Q}} = K(\vec{x})[(\!|p|\!)_{\vec{Q}}[\vec{S} \mapsto \vec{s}]\backslash \vec{s}] \qquad \text{where} \quad K \text{ non-atomic, and } \vec{s} \text{ fresh}$$

where the nodes $v^0, \ldots, v^{m-1}$ have special controls not present in $\mathscr{K}$, and they are used only to simplify the translation. In practice, these special nodes give a "name" to each hole of the bigraphs, i.e., the node v^i represents the i-hole of the bigraphs. Notice that, the hole sequence may not follow necessarily the numeration of holes, as shown in the bigraph in Figure 6.

The following result, states the expressivity of our language.

Proposition 6 *Let $G : \langle m, \vec{X}_B, (\bigcup \vec{X}_B) \uplus X_F \rangle \to \langle n, \vec{Y}_B, (\bigcup \vec{Y}_B) \uplus Y_F \rangle$ be a binding bigraph. Then, $(\!|G|\!)_{\vec{Q}}$ is an agent-graph s.t. $\vec{Q} : \vec{X}_B; X_F \vdash (\!|G|\!)_{\vec{Q}} : (\vec{Y}_B, Y_F)$, and $[\![\vec{Q} : \vec{X}_B; X_F \vdash (\!|G|\!)_{\vec{Q}} : (\vec{Y}_B, Y_F)]\!] = G$.*

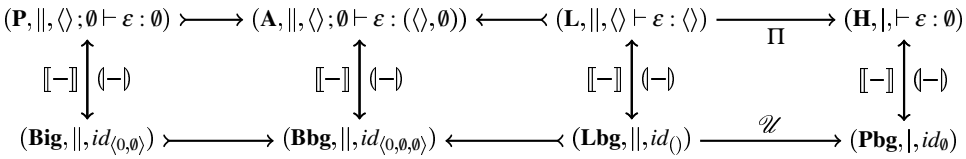


Figure 7: Relations among the categories under investigation.

We can also establish nice connections between the axiomatizations of the two categories.

Proposition 7 *Let $G, G' : \langle m, \vec{X}_B, (\bigcup \vec{X}_B) \uplus X_F \rangle \to \langle n, \vec{Y}_B, (\bigcup \vec{Y}_B) \uplus Y_F \rangle$ be two binding bigraphs over a given signature. Then, $G = G'$ if and only if $\vec{Q} : \vec{X}_B; X_F \vdash (\!|G|\!)_{\vec{Q}} \equiv (\!|G'|\!)_{\vec{Q}}$.*

Proposition 8 *For $\Gamma; \tau \vdash A : (\vec{\sigma}, \rho)$ a typing judgment: $\Gamma; \tau \vdash (\!|[\![\Gamma; \tau \vdash A : (\vec{\sigma}, \rho)]\!]|\!)_{\mathrm{dom}(\Gamma)} \equiv A$.*

These results induces a *normal form* for agent-graphs inspired to the discrete normal form of binding bigraphs. This normal form tries to separate the notions of nesting and linking:

$$A \equiv \nu \vec{z}. \left((\bar{A}_0 \parallel \ldots \parallel \bar{A}_{n-1})[\vec{X} \mapsto \vec{x}] \right)$$

$$\bar{A} \equiv \big(L_0(\vec{x}_0)[\bar{A}^0[\vec{Y}_0 \mapsto \vec{y}_0] \backslash \vec{y}_0] \mid \ldots \mid L_{m-1}(\vec{x}_{m-1})[\bar{A}^{m-1}[\vec{Y}_{m-1} \mapsto \vec{y}_{m-1}] \backslash \vec{y}_{m-1}] \mid$$
$$l_0(\vec{z}_0) \mid \ldots \mid l_{k-1}(\vec{z}_{k-1}) \mid X_0[\vec{Z}_0 \mapsto \vec{z}_0] \upharpoonright \vec{z}_0 \mid \cdots \mid X_{h-1}[\vec{Z}_{h-1} \mapsto \vec{z}_{h-1}] \upharpoonright \vec{z}_{h-1} \big) \downharpoonright W .$$

Proposition 9 *Every agent-graph is structural equivalent to an agent-graph in normal form.*

Finally, notice that the mapping $(\!|-|\!) : \mathbf{Bbg}(\mathscr{K}) \to \mathbf{A}(\mathscr{L}_{\mathscr{K}})$ is not a functor, because the composition of wirings in binding bigraphs is not respected by the graph composition defined in virtue of Lemma 1. Therefore, $\mathbf{Bbg}(\mathscr{K})$ and $\mathbf{A}(\mathscr{L}_{\mathscr{K}})$ are not isomorphic. However, as we will see next, composition is respected in the important subcategories of pure and local bigraphs.

5 Characterize pure, local and prime bigraphs

In this section we show that pure, local and prime bigraphs can be captured by simple syntactic conditions on the language and types of the typed language presented in Section 3. Indeed, these subcategories are covered by the same sublanguage, obtained by removing $\downharpoonright$ and $\upharpoonright$:

$$A ::= \varepsilon \mid \mathbf{0} \mid l(\vec{x}) \mid L(\vec{x})[A \backslash \vec{y}] \mid X \mid A | A \mid A \parallel A \mid \nu z.A \mid A[w \mapsto z] . \tag{1}$$

Despite we use the same (sub)language, and using essentially the same typing rules of Figure 3, we are able to describe both pure and local bigraphs, just by restricting the form of types and typing environment. A summary diagram of the correspondence among the categories under investigation is in Figure 7.

5.1 Pure bigraphs

In pure bigraphs all names are global, hence, variables and agents cannot have localized names. Therefore, a typing system for pure bigraphs is derivable from the system in Figure 3 by simply

$$\frac{}{\langle\rangle;\tau\vdash\varepsilon:(0,\tau)} \qquad \frac{}{\langle\rangle;\tau\vdash\mathbf{0}:(1,\tau)} \qquad \frac{}{\langle\rangle;\tau\vdash l(\vec{x}):(1,\vec{x}\cup\tau)}$$

$$\frac{\Gamma;\tau\vdash A:(1,\rho)}{\Gamma;\tau\vdash L(\vec{x})[A\backslash\langle\rangle]:(1,\rho\cup\vec{x})} \qquad \frac{}{X;\tau\vdash X:(1,\tau)}$$

$$\frac{\Gamma;\tau\vdash A:(1,\rho)\ \Gamma';\tau'\vdash A':(1,\rho')\ \tau\cap\tau'=\emptyset}{\Gamma,\Gamma';\tau\cup\tau'\vdash A\mid A':(1,\rho\cup\rho')} \qquad \frac{\Gamma;\tau\vdash A:(n,\rho)\ \Gamma';\tau'\vdash A':(n',\rho')\ \tau\cap\tau'=\emptyset}{\Gamma,\Gamma';\tau\cup\tau'\vdash A\mid\mid A':(n+n',\rho\cup\rho')}$$

$$\frac{\Gamma;\tau\vdash A:(n,\rho)}{\Gamma;\tau\vdash\nu x.A:(n,\rho\setminus\{x\})} \qquad \frac{\Gamma;\tau\vdash A:(n,\rho)\ \pi\text{ permutation}}{\pi(\Gamma);\tau\vdash A:(n,\rho)} \qquad \frac{\Gamma;\tau\vdash A:(n,\rho)}{\Gamma;\tau\vdash A[x\mapsto y]:(n,(\rho\setminus\{x\})\cup\{y\})}$$

Figure 8: Typing rules for restricted agent-graphs where all names are global.

restricting to types of the form $(\vec{\emptyset},\tau)$, while the variables in the environment can have only $\emptyset$ as type. The only function of $\vec{\emptyset}$ is to count the locations of the system. Therefore a typing judgement is simply of the form $\Gamma;\tau\vdash A:(n,\rho)$ where A is a term as per (1); a *global type* (n,ρ) is a pair where $n\in\mathbb{N}$ and ρ is a simple types; an environment $\Gamma;\tau$ is a list of variables $\Gamma=\vec{X}=X_0,\ldots,X_{n-1}$, together with a simple-type τ.

Notice that for L non-atomic, it must be $in(L)=0$, because there are no local names which can be linked to an in-tentacle. This is enforced by the typing system, which is given Figure 8. These rules are essentially the same of Figure 3, just with the restricted form of types and environments.

Definition 8 The category $\mathbf{P}(\mathscr{L})$ of agent-graphs, over a ranked alphabet $\mathscr{L}$, has types (m,ρ) as objects, and judgments as morphisms, i.e., if $X_0,\ldots,X_{n-1};\tau\vdash A:(m,\rho)$ then $(X_0,\ldots,X_{n-1},A):(n,\tau)\to(m,\rho)$ is a morphism. Composition is defined in virtue of Lemma 1.

Proposition 10 $(\mathbf{P}(\mathscr{L}),||,\langle\rangle;\emptyset\vdash\varepsilon:\emptyset)$ *is a strict symmetric monoidal category.*

The encoding functor $[\![-]\!]:\mathbf{Big}(\mathscr{K})\to\mathbf{L}(\mathscr{P}_{\mathscr{K}})$ and the representation function $(\!|-|\!):\mathbf{P}(\mathscr{K})\to\mathbf{Big}(\mathscr{K}_{\mathscr{L}})$ are particular cases of the ones for binding bigraphs. Again the two maps establish a bijection between the two categories.

5.2 Local bigraphs

In local bigraphs all names are localized, hence there are no global names, and variables can have only their own names. So, the typing is obtained again from the system in Figure 3 by simply restricting to types of the form $(\vec{\sigma},\emptyset)$, while in the environment the set of the global names is always $\emptyset$. More formally, a typing judgement is of the form $\Gamma\vdash A:\vec{\sigma}$ where A is a term generated by the grammar (1), a *local type* $\vec{\tau}=\tau_0\ldots\tau_{n-1}$ is a list of simple types, and an *environment* Γ is a list of typed variables $(\Gamma=\vec{X}:\vec{\tau}=X_0:\tau_0,\ldots,X_{n-1}:\tau_{n-1})$. The type inference rules are in Figure 9. Notice that, in local bigraphs, non-atomic hyperedges can have non-zero in-rank.

Definition 9 The category $\mathbf{L}(\mathscr{L})$ of agent-graphs, over a ranked alphabet $\mathscr{L}$, has local types $\vec{\sigma}$ as objects, and judgments as morphisms, that is, if $X_0:\tau_0,\ldots,X_{n-1}:\tau_{n-1}\vdash A:\vec{\sigma}$ then $(X_0,\ldots,X_{n-1},A):\vec{\tau}\to\vec{\sigma}$ is a morphism. Composition is defined in virtue of Lemma 1.

$$\frac{}{\langle\rangle \vdash \varepsilon : \langle\rangle} \qquad \frac{}{\langle\rangle \vdash \mathbf{0} : \emptyset} \qquad \frac{}{\langle\rangle \vdash l(\vec{x}) : \vec{x}} \qquad \frac{\Gamma \vdash A : \sigma \cup \vec{y} \quad \vec{y} \cap \sigma = \emptyset}{\Gamma \vdash L(\vec{x})[A \backslash \vec{y}] : \sigma \cup \vec{x}}$$

$$\frac{}{X : \sigma \vdash X : \sigma} \qquad \frac{\Gamma \vdash A : \sigma \quad \Gamma' \vdash A' : \sigma'}{\Gamma, \Gamma' \vdash A \mid A' : \sigma \cup \sigma'} \qquad \frac{\Gamma \vdash A : \vec{\sigma} \quad \Gamma' \vdash A' : \vec{\sigma}'}{\Gamma, \Gamma' \vdash A \parallel A' : \vec{\sigma}\vec{\sigma}'}$$

$$\frac{\Gamma \vdash A : \vec{\sigma}}{\Gamma \vdash \nu x.A : \vec{\sigma} \setminus \{x\}} \qquad \frac{\Gamma \vdash A : \vec{\sigma} \quad \pi \text{ permutation}}{\pi(\Gamma) \vdash A : \vec{\sigma}} \qquad \frac{\Gamma \vdash A : \vec{\sigma}}{\Gamma \vdash A[x \mapsto y] : (\vec{\sigma} \setminus \{x\}) \cup \{y\}}$$

Figure 9: Typing rules for restricted agent-graphs where all names are local.

$$\frac{}{\langle\rangle \vdash \mathbf{0} : \emptyset} \qquad \frac{}{\langle\rangle \vdash l(\vec{x}) : \vec{x}} \qquad \frac{\Gamma \vdash A : \sigma \quad \Gamma' \vdash A' : \sigma' \quad |\Gamma, \Gamma'| < 2}{\Gamma, \Gamma' \vdash A \mid A' : \sigma \cup \sigma'}$$

$$\frac{}{X : \sigma \vdash X : \sigma} \qquad \frac{\Gamma \vdash A : \sigma}{\Gamma \vdash \nu x.A : \sigma \setminus \{x\}} \qquad \frac{\Gamma \vdash A : \sigma}{\Gamma \vdash A[x \mapsto y] : (\sigma \setminus \{x\}) \cup \{y\}}$$

Figure 10: Typing rules for restricted agent-graphs with one locality.

Proposition 11 $(\mathbf{L}(\mathscr{L}), \|, \langle\rangle \vdash \varepsilon : \langle\rangle)$ *is a strict symmetric monoidal category.*

The two encoding functors $[\![-]\!] : \mathbf{Lbg}(\mathscr{K}) \to \mathbf{L}(\mathscr{L}_{\mathscr{K}})$, and $(\!|-|\!) : \mathbf{L}(\mathscr{K}) \to \mathbf{Lbg}(\mathscr{K}_{\mathscr{L}})$ are particular cases of the ones for binding bigraphs. Notice that, in this particular case, $(\!|-|\!)$ is actually a functor; and, as before, the two functors establish a bijection between the two categories.

5.3 Prime bigraphs

Following the idea of the functor $[\![-]\!]$ from agent-graph to bigraphs, we can identify a subcategory of **A**, where all agents have zero or one variable. These are *prime bigraph*, that is bigraphs with at most one hole. One may think of these bigraphs as single-located (open) systems.

Again we can characterize pure prime bigraphs by a restriction on agent types. A typing judgement is of the form $\Gamma \vdash A : \sigma$ where A is zero or one variable term generated by $A ::= \mathbf{0} \mid l(\vec{x}) \mid X \mid A|A \mid \nu z.A \mid A[w \mapsto z]$. A *prime type* σ is just a simple type, and an environment Γ is a list of typed variables of at most length one, i.e., $\Gamma ::= \langle\rangle \mid X : \rho$. The induced type inference rules are in Figure 10.

Definition 10 The category **H** has simple types (τ) as objects, and judgments as morphisms, i.e., if $X : \tau \vdash A : \rho$ then $(X, A) : \tau \to \rho$ is a morphism or if $\langle\rangle \vdash A : \rho$ then $(\langle\rangle, A) : \emptyset \to \rho$ is a morphism. Composition is defined as follows:

$$\frac{\Gamma \vdash A : \tau \quad X : \rho \vdash A' : \rho}{\Gamma \vdash A'\{A/X\} : \rho}$$

Proposition 12 $(\mathbf{H}, |, \langle\rangle \vdash \mathbf{0} : \emptyset)$ *is a strict symmetric monoidal category.*

The two encoding functors $[\![-]\!] : \mathbf{Pbg}(\mathscr{K}) \to \mathbf{H}(\mathscr{L}_{\mathscr{K}})$ and $(\!|-|\!) : \mathbf{H}(\mathscr{L}) \to \mathbf{Pbg}(\mathscr{L}_{\mathscr{K}})$ can be defined as a "simplification" of the ones for local bigraphs.

Proposition 13 *Let A, A' be terms; then, for every environment $X : \tau$: $X : \tau \vdash A \equiv A'$ if and only if $[\![X : \tau \vdash A : \tau']\!] = [\![X : \tau \vdash A' : \tau']\!]$.*

Proposition 14 *Let $H, H' : (\langle\rangle) \to (Y)$ be two prime graphs; then, $H = H'$ iff $\langle\rangle \vdash (\!|H|\!)_{\langle\rangle} \equiv (\!|H'|\!)_{\langle\rangle}$. Instead, if $H, H' : (X) \to (Y)$, then $H = H'$ iff $Q : X \vdash (\!|H|\!)_Q \equiv (\!|H'|\!)_Q$.*

Forgetting localities. Let us consider only atomic signatures $\mathscr{K}_a$, that is, where all controls are atomic, and hence there is no nesting of nodes. In this case, we can define a functor $\mathscr{U} : \mathbf{Lbg}(\mathscr{K}_a) \to \mathbf{Pbg}(\mathscr{K}_a)$ which "forgets" the localities of a local bigraph, merging all roots into a single one and all sites (holes) into a single one. Formally:
Objects: $\mathscr{U}((X_0 \dots X_{n-1})) = X_0 + \dots + X_{n-1}$.
Morphisms: $\mathscr{U}((V, E, ctrl, prnt, link)) = (V, E, ctrl, prnt', link)$ where $prnt'(v) = 0$, for all v.

The previous functors $[\![-]\!], (\!|-|\!)$ and the forgetful functor $\mathscr{U}$ induce a forgetful functor $\Pi : \mathbf{L}(\mathscr{L}_{\mathscr{K}_a}) \to \mathbf{H}(\mathscr{L}_{\mathscr{K}_a})$, defined as follows:
Objects: $\Pi(\sigma_0 \dots \sigma_{n-1}) = \sigma_0 + \dots + \sigma_{n-1}$.
Morphisms: given a graph in normal form $A \equiv \nu\vec{z}.\Big((\bar{A}_0 \parallel \dots \parallel \bar{A}_{n-1})[\vec{X} \mapsto \vec{x}]\Big)$, where every subgraph $\bar{A}_i \equiv \Big(l_0^i(\vec{z}_0^i) \mid \dots \mid l_{k_i-1}^i(\vec{z}_{k_i-1}^i) \mid X_0^i[\vec{Z}_0^i \mapsto \vec{z}_0^i] \mid \dots \mid X_{h_i-1}^i[\vec{Z}_{h_i-1}^i \mapsto \vec{z}_{h_i-1}^i]\Big)$, then

$$\Pi(A) = \nu\vec{z}.\big((l_0^0(\vec{z}_0^0) \mid \dots \mid l_{k_0-1}^0(\vec{z}_{k_0-1}^0)) \mid \dots \mid (l_0^{n-1}(\vec{z}_0^{n-1}) \mid \dots \mid l_{k_{n-1}-1}^{n-1}(\vec{z}_{k_{n-1}-1}^{n-1})) \mid X[\vec{Z}_0^0 \mapsto \vec{z}_0^0] \dots [\vec{Z}_{h_0-1}^0 \mapsto \vec{z}_{h_0-1}^0] \dots [\vec{Z}_0^{n-1} \mapsto \vec{z}_0^{n-1}] \dots [\vec{Z}_{h_{n-1}-1}^{n-1} \mapsto \vec{z}_{h_{n-1}-1}^{n-1}]\big)$$

In practice the above functor merges all the separate agent-graphs into a single-located graph. It translates a $\parallel$ operator with the $\mid$ one and unifies all variables into a single one.

As a consequence of the definitions of the functors defined above, we can prove the following results. Notice that the lists $\vec{X}$, $\vec{Q}$ and $\vec{\tau}$ are either empty or just singletons.

Proposition 15 *Let $\vec{X} : \vec{\tau} \vdash A : \rho = \Pi(\Gamma \vdash B : \vec{\sigma})$; then, $\vec{X} : \vec{\tau} \vdash (\!|\mathscr{U}([\![\Gamma \vdash B : \vec{\sigma}]\!])|\!)_{\vec{X}} \equiv A$.*

Proposition 16 *1. Let $P : (\vec{X}) \to (Y)$ be a prime bigraph, then $[\![\vec{Q} : \vec{X} \vdash (\!|P|\!)_{\vec{Q}}]\!] = P$.*

2. Let $\vec{X} : \vec{\tau} \vdash A : \sigma$ be a term, then $\vec{X} : \vec{\tau} \vdash (\!|[\![\vec{X} : \vec{\tau} \vdash A : \sigma]\!]|\!)_{\vec{X}} \equiv A$.

6 Comparing with SHR hypergraphs and ADR designs

Remarkably, our language for binding bigraphs can be used for capturing formalisms introduced in literature, often for quite different purposes. Here we consider the hypergraphs used in *Synchronized Hyperedge Rewriting* (SHR) [FHL+05] and the "designs" of *Architectural Design Rewriting* (ADR) [BLMT07]. Both are derived from the algebra of graphs introduced first in [CMR94].

SHR hypergraphs. SHR is a framework that allows hypergraph transformations by means of local productions replacing a single hyperedge by a generic hypergraph, possibly with constraints given by the surrounding nodes. The global rewriting is obtained by combining different local production whose conditions are compatible (with respect to some synchronization model).

In this paper, we are interested only in SHR hypergraphs, which are inductively defined as:

$$G ::= \mathbf{0} \mid l(\vec{x}) \mid G|G \mid \nu x.G$$

where $\mathbf{0}$ is the empty graph, the hyperedge l is linked to the nodes in $\vec{x}$, and ν binds x in G.

Clearly, the SHR grammar is a particular case of the one for prime bigraphs (Section 5.3), and specifically when the variable and substitutions are dropped.

ADR designs. ADR graphs (called *designs*) resemble SHR graphs, but they have a notion of graph nesting, as some hyperedges can contain other graphs. Such nesting is used for *incremental modelling*, that is, edges can be refined into graphs or vice versa graphs collapse into edges. The ADR designs are inductively defined as:

$$D ::= L[\lambda\vec{x}.G] \qquad G ::= \mathbf{0} \mid x \mid l(\vec{x}) \mid G|G \mid \nu x.G \mid D(\vec{x})$$

where $\mathbf{0}$ is the empty graph, the hyperedge l is linked to the nodes in $\vec{x}$, ν binds x in G, $D(\vec{x})$ is a design generated by attaching design D to nodes $\vec{x}$, and finally $L[\lambda\vec{x}.G]$ represent a design L, with "body graph" G and exposing the names $\vec{x}$ in its interface.

The grammar of designs recalls the one defined for local bigraphs, when the $\|$ composition is omitted. In such a case, we deal with graphs residing in only one location. A formal translation of the ADR design grammar into the grammar in (1) can be defined as follow:

$$T(\mathbf{0}) = \mathbf{0} \qquad T(x) = \mathbf{0}[z \mapsto x] \quad (z \text{ fresh}) \qquad T(G_1 \mid G_2) = T(G_1) \mid T(G_2)$$
$$T(l(\vec{x})) = l(\vec{x}) \qquad T(\nu x.G) = \nu x.T(G) \qquad T(L[\lambda\vec{x}.G](\vec{y})) = L(\vec{y})[G\backslash\vec{x}]$$

By means of these translations of SHR hypergraphs as prime bigraphs, and ADR designs as local bigraphs, we can transfer results and constructions among formalisms developed independently. As examples, it is possible to extend the SHR semantics allowing for not only replacing single hyperedges, but more complex graphs; moreover, we can also define congruent bisimulations for SHR systems via the so-called IPO construction over bigraphical reactive systems.

7 Conclusion

In this paper we have first defined an algebra of typed term graphs which corresponds precisely to binding bigraphs, on a given signature. Secondly, we have shown that particular sublanguages of our main language properly characterize interesting subclasses of bigraphs, more precisely: pure and local bigraphs. Moreover, on this last kind of bigraphs we also give a reduced language for dealing with one-location (bi)graphs, named prime bigraphs. So, those languages can be used in place of the more complex bigraph algebra already present in literature. A family of bigraphical calculi has been introduced in [DK08]; however, these calculi has been suitably restricted for modelling biological systems and do not cover all possible bigraphs over a given signature.

Finally, it turns out that these languages are strictly connected with two well-know formalisms: *Synchronized Hyperedge Replacement* hypergraphs, which can be represented as a sublanguage of the algebra for prime bigraphs (over atomic signatures), and *Architectural Design Rewriting* designs, which are a sub-case of the local bigraphs' language.

A possible future work is to take advantage of the rich theory provided by bigraphical reactive systems [JM03], in order to obtain interesting results about SHR and ADR. In particular, we hope to generalize the transitions allowed in SHR graphs, which only rewrites a single hyperedge, to more general ones dealing with (sub)graphs. Moreover, bigraphs allow to synthesise labelled transition systems out of rewriting rules, via the so-called *idem-pushout* construction [LM00]; it is important to notice that the bisimilarity induced by this labelled transitions system (LTS) is always a congruence. Therefore, given a reactive system over SHR (ADR) graphs, we can derive the labelled transition system in bigraphs, and remap it on SHR (ADR) graphs. Then, the inductive definition of SHR (ADR) agents can be useful for defining an SOS-like presentation of the LTS derived in this way.

Acknowledgments. The authors thank Emilio Tuosto and Ivan Lanese for useful discussions about SHR.

Bibliography

[BDE+06] L. Birkedal, S. Debois, E. Elsborg, T. Hildebrandt, H. Niss. Bigraphical Models of Context-Aware Systems. In Aceto and Ingólfsdóttir (eds.), *Proc. FoSSaCS*. Lecture Notes in Computer Science 3921, pp. 187–201. Springer, 2006.

[BDGM07] L. Birkedal, T. C. Damgaard, A. J. Glenstrup, R. Milner. Matching of Bigraphs. *Electr. Notes Theor. Comput. Sci.* 175(4):3–19, 2007.

[BGH+08] M. Bundgaard, A. J. Glenstrup, T. T. Hildebrandt, E. Højsgaard, H. Niss. Formalizing Higher-Order Mobile Embedded Business Processes with Binding Bigraphs. In Lea and Zavattaro (eds.), *COORDINATION*. Lecture Notes in Computer Science 5052, pp. 83–99. Springer, 2008.

[BLMT07] R. Bruni, A. Lluch-Lafuente, U. Montanari, E. Tuosto. Service Oriented Architectural Design. In Barthe and Fournet (eds.), *Proc. TGC*. Lecture Notes in Computer Science 4912, pp. 186–203. Springer, 2007.

[CMR94] A. Corradini, U. Montanari, F. Rossi. An Abstract Machine for Concurrent Modular Systems: CHARM. *Theor. Comput. Sci.* 122(1&2):165–200, 1994.

[DB06] T. C. Damgaard, L. Birkedal. Axiomatizing Binding Bigraphs. *Nord. J. Comput.* 13(1-2):58–77, 2006.

[DK08] T. C. Damgaard, J. Krivine. A Generic Language for Biological Systems based on Bigraphs. Technical report TR-2008-115, IT University of Copenhagen, Dec. 2008.

[FHL+05] G. L. Ferrari, D. Hirsch, I. Lanese, U. Montanari, E. Tuosto. Synchronised Hyperedge Replacement as a Model for Service Oriented Computing. In Boer et al. (eds.), *Proc. FMCO*. Lecture Notes in Computer Science 4111, pp. 22–43. Springer, 2005.

[GM07] D. Grohmann, M. Miculan. Reactive Systems over Directed Bigraphs. In Caires and Vasconcelos (eds.), *Proc. CONCUR 2007*. Lecture Notes in Computer Science 4703, pp. 380–394. Springer, 2007.

[JM03] O. H. Jensen, R. Milner. Bigraphs and transitions. In *Proc. POPL*. Pp. 38–49. 2003.

[JM04] O. H. Jensen, R. Milner. Bigraphs and mobile processes (revised). Technical report UCAM-CL-TR-580, Computer Laboratory, University of Cambridge, 2004.

[LM00] J. J. Leifer, R. Milner. Deriving Bisimulation Congruences for Reactive Systems. In Palamidessi (ed.), *Proc. CONCUR*. Lecture Notes in Computer Science 1877, pp. 243–258. Springer, 2000.

[LM06] J. J. Leifer, R. Milner. Transition systems, link graphs and Petri nets. *Mathematical Structures in Computer Science* 16(6):989–1047, 2006.

[Mil01] R. Milner. Bigraphical Reactive Systems. In Larsen and Nielsen (eds.), *Proc. 12th CONCUR*. Lecture Notes in Computer Science 2154, pp. 16–35. Springer, 2001.

[Mil04] R. Milner. Bigraphs whose names have multiple locality. Technical report 603, University of Cambridge, CL, Sept. 2004.

A Structural congruence

The free name function fn is defined as follows with respect to an environment (Γ, τ).

$$
\begin{aligned}
fn_{\Gamma;\tau}(\varepsilon) &= \tau & \qquad fn_{\Gamma;\tau}(L(\vec{x})[A\backslash\vec{y}]) &= (fn_{\Gamma;\tau}(A)\setminus\vec{y})\cup\vec{x}\cup\tau \\
fn_{\Gamma;\tau}(\mathbf{0}) &= \tau & fn_{\Gamma;\tau}(A_1 \mid A_2) &= fn_{\Gamma;\tau}(A_1)\cup fn_{\Gamma;\tau}(A_2) \\
fn_{\Gamma;\tau}(l(\vec{x})) &= \vec{x}\cup\tau & fn_{\Gamma;\tau}(A_1 \parallel A_2) &= fn_{\Gamma;\tau}(A_1)\cup fn_{\Gamma;\tau}(A_2) \\
fn_{\Gamma;\tau}(\nu y.A) &= fn_{\Gamma;\tau}(A)\setminus\{y\} & fn_{\Gamma;\tau}(A[w\mapsto z]) &= (fn_{\Gamma;\tau}(A)\setminus\{w\})\cup\{z\} \\
fn_{\Gamma;\tau}(A \downarrow x) &= fn_{\Gamma;\tau}(A) & fn_{\Gamma;\tau}(A \upharpoonright x) &= fn_{\Gamma;\tau}(A) \\
fn_{\Gamma;\tau}(X_i) &= \tau_i\cup\tau \quad \text{if } X_i{:}\tau_i\in\Gamma
\end{aligned}
$$

In the following table, the structural congruence for agent-graph is defined with respect to some environment $\Gamma;\tau$.

$$\Gamma;\tau\vdash A\mid \mathbf{0}\equiv A$$
$$\Gamma;\tau\vdash A_1\mid A_2\equiv A_2\mid A_1$$
$$\Gamma;\tau\vdash (A_1\mid A_2)\mid A_3\equiv A_1\mid (A_2\mid A_3)$$
$$\Gamma;\tau\vdash A\parallel \varepsilon\equiv A$$
$$\Gamma;\tau\vdash \varepsilon\parallel A\equiv A$$
$$\Gamma;\tau\vdash (A_1\parallel A_2)\parallel A_3\equiv A_1\parallel (A_2\parallel A_3)$$
$$\Gamma;\tau\vdash \nu x.\mathbf{0}\equiv \mathbf{0}\text{ if } x\notin fn_{\Gamma;\tau}(\mathbf{0})$$
$$\Gamma;\tau\vdash \nu x.\varepsilon\equiv \varepsilon\text{ if } x\notin fn_{\Gamma;\tau}(\varepsilon)$$
$$\Gamma;\tau\vdash \nu x.\nu y.A\equiv \nu y.\nu x.A$$
$$\Gamma;\tau\vdash \nu x.A\equiv \nu y.(A\{y/x\})\text{ if } y\notin fn_{\Gamma;\tau}(A)$$
$$\Gamma;\tau\vdash A[x\mapsto y]\equiv (A\{z/x\})[z\mapsto y]\text{ if } z\notin fn_{\Gamma;\tau}(A)$$
$$\Gamma;\tau\vdash A[x\mapsto x]\equiv A\text{ if } x\in fn_{\Gamma;\tau}(A)$$
$$\Gamma;\tau\vdash A[x\mapsto y]\equiv A\text{ if } x\notin fn_{\Gamma;\tau}(A)\wedge y\in fn_{\Gamma;\tau}(A)$$
$$\Gamma;\tau\vdash A[x\mapsto y][w\mapsto z]\equiv A[w\mapsto z][x\mapsto y]\text{ if } x\neq z,y\neq w$$
$$\Gamma;\tau\vdash A[x\mapsto y][y\mapsto z]\equiv A[x\mapsto z]\text{ if } y\notin fn_{\Gamma;\tau}(A)$$
$$\Gamma;\tau\vdash \nu y.(A[x\mapsto y])\equiv \nu x.A\text{ if } y\notin fn_{\Gamma;\tau}(A)$$
$$\Gamma;\tau\vdash \nu z.(A[x\mapsto y])\equiv (\nu z.A)[x\mapsto y]\text{ if } z\notin \{x,y\}$$
$$\Gamma;\tau\vdash A\downarrow x\equiv A\text{ if } x\in fn_{\Gamma;\tau}(A)$$
$$\Gamma;\tau\vdash A\uparrow x\equiv A\text{ if } x\in fn_{\Gamma;\tau}(A)$$
$$\Gamma;\tau\vdash A\downarrow x[x\mapsto y]\equiv A[x\mapsto y]\downarrow y$$
$$\Gamma;\tau\vdash A\downarrow x[y\mapsto z]\equiv A[y\mapsto z]\downarrow x\text{ if } y\neq x\wedge z\neq x$$
$$\Gamma;\tau\vdash A\uparrow x[x\mapsto y]\equiv A[x\mapsto y]\uparrow y$$
$$\Gamma;\tau\vdash A\uparrow x[y\mapsto z]\equiv A[y\mapsto z]\uparrow x\text{ if } y\neq x\wedge z\neq x$$
$$\Gamma;\tau\vdash \nu x.(A\downarrow x)\equiv \nu x.A\text{ if } x\in fn_{\Gamma;\tau}(A)$$
$$\Gamma;\tau\vdash \nu y.(A\downarrow x)\equiv (\nu y.A)\downarrow x\text{ if } x\neq y$$
$$\Gamma;\tau\vdash \nu x.(A\uparrow x)\equiv \nu x.A\text{ if } x\in fn_{\Gamma;\tau}(A)$$
$$\Gamma;\tau\vdash \nu y.(A\uparrow x)\equiv (\nu y.A)\uparrow x\text{ if } x\neq y$$

$$\Gamma;\tau\vdash \nu x.(A_1\mid A_2)\equiv \nu x.A_1\mid A_2\text{ if } x\notin fn_{\Gamma;\tau}(A_2)$$
$$\Gamma;\tau\vdash \nu x.(A_1\parallel A_2)\equiv \nu x.A_1\parallel A_2\text{ if } x\notin fn_{\Gamma;\tau}(A_2)$$
$$\Gamma;\tau\vdash \nu x.(A_1\parallel A_2)\equiv A_1\parallel \nu x.A_2\text{ if } x\notin fn_{\Gamma;\tau}(A_1)$$
$$\Gamma;\tau\vdash (A_1\mid A_2)[x\mapsto y]\equiv A_1[x\mapsto y]\mid A_2\text{ if } x\notin fn_{\Gamma;\tau}(A_2)$$
$$\Gamma;\tau\vdash (A_1\mid A_2)[x\mapsto y]\equiv A_1[x\mapsto y]\mid A_2[x\mapsto y]$$
$$\Gamma;\tau\vdash (A_1\parallel A_2)[x\mapsto y]\equiv A_1[x\mapsto y]\parallel A_2\text{ if } x\notin fn_{\Gamma;\tau}(A_2)\wedge x\in fn_{\Gamma;\tau}(A_1)$$
$$\Gamma;\tau\vdash (A_1\parallel A_2)[x\mapsto y]\equiv A_1\parallel A_2[x\mapsto y]\text{ if } x\notin fn_{\Gamma;\tau}(A_1)\wedge x\in fn_{\Gamma;\tau}(A_2)$$
$$\Gamma;\tau\vdash (A_1\parallel A_2)[x\mapsto y]\equiv A_1[x\mapsto y]\parallel A_2[x\mapsto y]\text{ if } x\in fn_{\Gamma;\tau}(A_1)\cap fn_{\Gamma;\tau}(A_2)$$
$$\Gamma;\tau\vdash (A_1\parallel A_2)[x\mapsto y]\equiv A_1[x\mapsto y]\parallel A_2[x\mapsto y]\text{ if } x\notin fn_{\Gamma;\tau}(A_1)\cup fn_{\Gamma;\tau}(A_2)$$
$$\Gamma;\tau\vdash (A_1\mid A_2)\downarrow x\equiv A_1\downarrow x\mid A_2\text{ if } x\in fn_{\Gamma;\tau}(A_1)\wedge x\notin fn_{\Gamma;\tau}(A_2)$$
$$\Gamma;\tau\vdash (A_1\mid A_2)\downarrow x\equiv A_1\downarrow x\mid A_2\downarrow x\text{ if } x\in fn_{\Gamma;\tau}(A_1)\cap fn_{\Gamma;\tau}(A_2)$$
$$\Gamma;\tau\vdash (A_1\mid A_2)\uparrow x\equiv A_1\uparrow x\mid A_2\text{ if } x\in fn_{\Gamma;\tau}(A_1)\wedge x\notin fn_{\Gamma;\tau}(A_2)$$
$$\Gamma;\tau\vdash (A_1\mid A_2)\uparrow x\equiv A_1\uparrow x\mid A_2\uparrow x\text{ if } x\in fn_{\Gamma;\tau}(A_1)\cap fn_{\Gamma;\tau}(A_2)$$
$$\Gamma;\tau\vdash (A_1\parallel A_2)\uparrow x\equiv A_1\uparrow x\parallel A_2\text{ if } x\in fn_{\Gamma;\tau}(A_1)\wedge x\notin fn_{\Gamma;\tau}(A_2)$$
$$\Gamma;\tau\vdash (A_1\parallel A_2)\uparrow x\equiv A_1\uparrow x\parallel A_2\uparrow x\text{ if } x\notin fn_{\Gamma;\tau}(A_1)\wedge x\in fn_{\Gamma;\tau}(A_2)$$
$$\Gamma;\tau\vdash (A_1\parallel A_2)\uparrow x\equiv A_1\uparrow x\parallel A_2\uparrow x\text{ if } x\in fn_{\Gamma;\tau}(A_1)\cap fn_{\Gamma;\tau}(A_2)$$
$$\Gamma;\tau\vdash l(\vec{x})[x\mapsto z]\equiv l(\vec{x})\{z/x\}\text{ if } x\in\vec{x}$$
$$\Gamma;\tau\vdash L(\vec{x})[A\backslash\vec{y}]\equiv L(\vec{x})[(A\{z/y\})\backslash(\vec{y}\{z/y\})]\text{ if } x\in\vec{y}\wedge z\notin\vec{y}\cup fn_\Gamma(A)$$
$$\Gamma;\tau\vdash \nu z.L(\vec{x})[A\backslash\vec{y}]\equiv L(\vec{x})[(\nu z.A)\backslash\vec{y}]\text{ if } z\notin\vec{x}\cup\vec{y}$$
$$\Gamma;\tau\vdash L(\vec{x})[A\backslash\vec{Y}][x\mapsto w]\equiv L(\vec{x})\{w/x\})[A\backslash\vec{y}]\text{ if } x\in\vec{x}\wedge w\notin(fn_\Gamma(A)\setminus\vec{y})$$
$$\Gamma;\tau\vdash L(\vec{x})[A\backslash\vec{y}][w\mapsto z]\equiv L(\vec{x})[(A[w\mapsto z])\backslash\vec{y}]\text{ if } w\notin\vec{x}\cup\vec{y}\wedge z\notin\vec{y}$$
$$\Gamma;\tau\vdash L(\vec{x})[A[w\mapsto y]\backslash\vec{y}]\equiv L(\vec{x})[A\backslash(\vec{y}\{w/y\})]\text{ if } y\in\vec{y}\wedge w\notin fn_{\Gamma;\tau}(A)$$
$$\Gamma;\tau\vdash L(\vec{x})[A\backslash\vec{y}]\downarrow z\equiv L(\vec{x})[(A\downarrow z)\backslash\vec{y}]\text{ if } z\notin\vec{x}\cup\vec{y}$$
$$\Gamma;\tau\vdash L(\vec{x})[A\backslash\vec{y}]\uparrow z\equiv L(\vec{x})[(A\uparrow z)\backslash\vec{y}]\text{ if } z\notin\vec{x}\cup\vec{y}$$

Electronic Communications of the EASST
Volume of the Pre-proceedings of GT-VMT (2010)

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

Recognizable Graph Languages for Checking Invariants

Christoph Blume, H.J. Sander Bruggink and Barbara König

13 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

Recognizable Graph Languages for Checking Invariants

Christoph Blume, H.J. Sander Bruggink and Barbara König

Abteilung für Informatik und Angewandte Kognitionswissenschaft,
Universität Duisburg-Essen, Germany
christoph.blume@uni-due.de, sander.bruggink@uni-due.de, barbara_koenig@uni-due.de

Abstract: We generalize the order-theoretic variant of the Myhill-Nerode theorem to graph languages, and characterize the recognizable graph languages as the class of languages for which the Myhill-Nerode quasi order is a well quasi order. In the second part of the paper we restrict our attention to graphs of bounded interface size, and use Myhill-Nerode quasi orders to verify that, for such bounded graphs, a recognizable graph property is an invariant of a graph transformation system. A recognizable graph property is a recognizable graph language, given as an automaton functor. Finally, we present an algorithm to approximate the Myhill-Nerode ordering.

Keywords: graph transformation, recognizable graph languages, Myhill-Nerode theorem, invariants

1 Introduction

Regular languages and well quasi orders have proven to be useful analysis techniques in the field of string rewrite systems. In particular, the Myhill-Nerode well quasi order of a regular language L, which is strongly related to the well-known Myhill-Nerode equivalence, has nice properties [EHR83, LV94]: the left and right concatenation are monotone w.r.t. the order and the regular language L used to define it is upward-closed with respect to it. Let a string rewrite system $\mathscr{S}$ be given. From the first property it follows that if r is greater (with respect to the order) than ℓ for every rewrite rule $\ell \to r$ of $\mathscr{S}$, then it holds that v is greater than w for each word v reachable from w. The second property means, that for each word v that is greater than w, it holds that $v \in L$ if $w \in L$. Together, these two properties ensure that it is decidable whether a property, described as a regular language containing exactly the words satisfying the property, is an invariant of a string rewrite system.

Since the late 1980s several notions of regular graph languages – in this context called *recognizable* graph languages – have been introduced [BC87, Cou90, BK06, BK08b], which all turned out to be equivalent. Recognizable graph languages have found many applications, especially in the field of complexity theory.

In the light of the above observations it is natural to ask how results from regular languages, such as Myhill-Nerode equivalences, can be transferred and used for recognizable graph languages. While Myhill-Nerode equivalences are typically used to show that a language is not regular, we use them in a different way and study Myhill-Nerode quasi orders in order to verify that a specified property is an invariant of a graph transformation system.

The definition of recognizable graph language we use in this paper is based on the notion of automaton functor introduced in [BK08b], a category-based generalization of finite (word)

automata. Like finite automata in the word case, automaton functors provide an operational view on recognizable graph languages, which allows one to define a "Myhill-Nerode"-order on automaton states rather than on graphs directly. This is convenient, because states typically represent an infinite class of graphs. Still, automaton functors are in general infinite structures, due to the unboundedness of graph interfaces. In Section 2 we briefly define recognizable graph languages, automaton functors, and the category-theoretic notions at the heart thereof.

In Section 3 we generalize the order-theoretic variant of the Myhill-Nerode theorem to (recognizable) graph languages; that is, we define the Myhill-Nerode quasi order on graph languages and characterize recognizable graph languages as the class of languages for which this order is a well quasi order.

In the second part of the paper we focus on the application of the Myhill-Nerode quasi order in practice. First, in Section 4 we show that we need only define the automaton functor for a restricted set of so-called *atomic cospans*, so that we do not need consider *all* cospans when calculating the order.

As indicated above, the quasi order typically cannot be represented in a finite way, due to the unboundedness of graph interfaces. In Section 5 therefore, we restrict our attention to graphs which can be constructed with atomic cospans of bounded interface sizes, and we present an algorithm which approximates (and in the case of deterministic automaton functors even computes) the Myhill-Nerode quasi order of an automaton functor. Finally, we illustrate the work with a short example in Section 6. The full version with the proofs can be found at http://www.ti.inf.uni-due.de/publications/blume/invcheck.pdf

2 Preliminaries

In this section we briefly recall some concepts of category theory and recognizable graph languages. We presuppose a basic knowledge of category theory and order theory.

2.1 Category Theory and Recognizable Graph Languages

First we review and fix some notations. The category which has sets as objects, relations as arrows and relation composition as composition operator is denoted by $\mathcal{Rel}$. The subcategory which has total functions as arrows instead of relations is denoted by $\mathcal{Set}$. The composition of two arrows f and g will be denoted by ; where $f;g = g \circ f$ indicates the arrow which is obtained by first applying the arrow f and then the arrow g.

Let $\mathscr{C}$ be a category with pushouts. A cospan $c\colon J \xrightarrow{c^L} C \xleftarrow{c^R} K$ is a pair of $\mathscr{C}$-arrows with the same codomain. Here, J and K are the domain (or *inner interface*) and codomain (or *outer interface*) of the cospan c, respectively. The identity cospan for an object E is the cospan consisting of twice the identity arrow of E. Let $c\colon J \xrightarrow{c^L} C \xleftarrow{c^R} K$ and $d\colon K \xrightarrow{d^L} D \xleftarrow{d^R} M$ be cospans (where the codomain of c equals the domain of d). The composition of c and d is obtained by taking the pushout of c^R and d^L. A *semi-abstract cospan* is an equivalence class of cospans, where we take the middle object of the cospan up to isomorphism. Now, the cospan category $Cospan(\mathscr{C})$ is defined as the category which has the objects of $\mathscr{C}$ as objects, and semi-abstract cospans as arrows. If the middle object is not important, a cospan $c\colon J \to C \leftarrow K$ (an arrow in the

cospan category from J to K) will be denoted as $c\colon J \nrightarrow K$.

Let a set Σ of labels be given. A *hypergraph* G, later also simply called *graph*, is a four-tuple $\langle V_G, E_G, \mathsf{att}_G, \mathsf{lab}_G\rangle$, where V_G is a finite set of *vertices* (or *nodes*) of G, E_G is a finite set of *edges* of G, $\mathsf{att}_G\colon E_G \to V_G^*$ is the *attachment function* and $\mathsf{lab}_G\colon E_G \to \Sigma$ is the *labeling function*. Here, V_G^* denotes the set of finite sequences of elements of V_G. A *hypergraph morphism* f is a structure-preserving map between two hypergraphs. A discrete graph is a graph which does not contain any edges. The discrete graph with n nodes is denoted by D_n. The *empty graph* is denoted by $\emptyset$ instead of D_0. The category of graphs and graph morphisms is denoted by $\mathcal{HG}raph$.

A cospan of graphs (an arrow in the category $Cospan(\mathcal{HG}raph)$) can be seen as a graph with an inner (left) and an outer (right) interface. Intuitively, the interfaces designate the parts of the graph which can be "touched" from the outside. With $[G]\colon \emptyset \to G \leftarrow \emptyset$ we denote the cospan consisting of a graph G with empty inner and outer interfaces.

Cospans of graphs are closely related to graph transformation systems, in particular to the double-pushout (DPO) approach to graph rewriting [SS05]. A DPO rewrite rule $\rho\colon L \leftarrow \rho_{\mathsf{L}} - I - \rho_{\mathsf{R}} \rightarrow R$ can be considered as a pair of cospans $\ell\colon \emptyset \to L \leftarrow \rho_{\mathsf{L}} - I$ and $r\colon \emptyset \to R \leftarrow \rho_{\mathsf{R}} - I$, which will in the following be called left- and right-hand side, respectively. Then it holds that $G \Rightarrow_\rho H$ if and only if $[G] = \ell ; c$ and $[H] = r ; c$, for some cospan c.

We define recognizable graph languages by using automaton functors on the category of cospans of graphs, as in [BK08b].

Definition 1 (Automaton functor, recognizability) Let a category $\mathscr{C}$ with initial object $\emptyset$ be given. An automaton functor is a functor $\mathscr{A}\colon \mathscr{C} \to \mathcal{R}el$, which maps every object X of $\mathscr{C}$ to a finite set $\mathscr{A}(X)$ of *states* of X and every arrow $f\colon X \to Y$ to a relation $\mathscr{A}(f) \subseteq \mathscr{A}(X) \times \mathscr{A}(Y)$, together with two distinguished sets $\mathrm{I}^{\mathscr{A}} \subseteq \mathscr{A}(\emptyset)$ and $\mathrm{F}^{\mathscr{A}} \subseteq \mathscr{A}(\emptyset)$ of *initial* and *final states*, respectively.

An automaton functor is *deterministic* if every relation $\mathscr{A}(f)$ is a function and every $\mathrm{I}^{\mathscr{A}}$ contains exactly one element.

An arrow $f\colon \emptyset \to \emptyset$ is accepted by an automaton functor $\mathscr{A}$, if $\langle s,t\rangle \in \mathscr{A}(f)$, for some $s \in \mathrm{I}^{\mathscr{A}}$ and $t \in \mathrm{F}^{\mathscr{A}}$. The language $L(\mathscr{A})$ of an automaton functor contains exactly those arrows which are accepted by it. A language L of arrows from $\emptyset$ to $\emptyset$ is a *recognizable language* if $L = L(\mathscr{A})$, for some automaton functor $\mathscr{A}$.

The intuition behind the definition is to have a mapping into a (locally) finite domain. The functor property guarantees that decomposing an object in different ways does not affect acceptance in any way. This is different from word languages, where there is essentially one way to decompose an object into subobjects.

Familiar constructions on finite automata, such as the determinization construction, can be easily generalized to automaton functors. Also, it was shown in [BK08b], that restricting to discrete interfaces does not affect the expressiveness of the formalism. Due to the latter result, we shall restrict to discrete interfaces in the rest of this paper.

The above definition can easily be generalized to accept languages between arbitrary objects. However, in our setting we require only languages from the initial object to the initial object.

A characterization of recognizable graph languages can be obtained in terms of recognizable languages in $Cospan(\mathcal{HG}raph)$:

Definition 2 (Recognizable graph language) A set L of graphs is a *recognizable graph language*, if $[L] = \{[G]\colon \emptyset \to G \leftarrow \emptyset \mid G \in L\}$ is a recognizable language in $Cospan(\mathcal{HG}raph)$.

In the following we will not distinguish between L, a language of graphs, and $[L]$, a language of (cospans of) graphs with empty interfaces.

2.2 Orders on categories

One of the basic concepts in checking invariants of regular languages is the notion of (well) quasi orders. First, we review the definition of (well) quasi orders on arbitrary sets (see also [LV94]).

A *quasi order (qo)* is a binary relation $\sqsubseteq_M$ on a set M if $\sqsubseteq_M$ is *reflexive* and *transitive*. A quasi order $\sqsubseteq_M$ on M is called *well-quasi order (wqo)* whenever if $m_1, m_2, \ldots$ is an infinite sequence of elements of M, then there exist integers i, j such that $0 < i < j$ and $m_i \sqsubseteq m_j$. In the following we will write $\sqsubseteq$ instead of $\sqsubseteq_M$ if M is clear from the context.

Next, we consider a semigroup $(M, *)$ and a quasi order $\sqsubseteq$ on M. We say that $\sqsubseteq$ is *left-monotone* (resp. *right-monotone*) if for all $m_1, m_2, m \in M$ the following condition is satisfied:

$$m_1 \sqsubseteq m_2 \implies m * m_1 \sqsubseteq m * m_2 \quad (\text{resp.} \quad m_1 \sqsubseteq m_2 \implies m_1 * m \sqsubseteq m_2 * m).$$

In the following we will define orders on the homsets of a category. More specifically, two arrows f, g can only be related by a quasi order $\sqsubseteq$ if they have the same source and target objects. Alternatively we could consider $\sqsubseteq$ as a family of quasi orders, one for each homset.

The notion of order in categories is also present in enriched categories [GMM94, Kel82]. Note however that unlike in enriched categories we do not necessarily require that the order is always preserved by composition ($f \sqsubseteq f'$ and $g \sqsubseteq g'$ implies $f;g \sqsubseteq f';g'$), since we will usually only require right-monotonicity as defined above.

3 A Generalization of the Myhill-Nerode Theorem

In this section we generalize the theorem of Myhill-Nerode to graph languages. This theorem says that a language is regular if and only if it is the union of equivalence classes of a monotone (or right-monotone) congruence on words of finite index. There is an order-theoretic variant of this theorem given in [EHR83, LV94] saying that a language is regular if and only if it is upward-closed with respect to a monotone well quasi order.

In order to state this theorem in our framework we first need the notion of Myhill-Nerode quasi order. Note that while the word or string variant of this theorem uses orders that are both left-monotone and right-monotone, here we work only with right-monotone orders. Intuitively this is sufficient since we start with the empty interface and attaching any cospan on the left can always be simulated by attaching an appropriate cospan on the right.

Definition 3 (Myhill-Nerode quasi order) Let L be a graph language over $Cospan(\mathcal{HG}raph)$. A quasi order $\leq_L$ on $Cospan(\mathcal{HG}raph)$ is called *Myhill-Nerode quasi order (relative to L)*, if for arbitrary cospans $a, b\colon \emptyset \nrightarrow D_n$ the following condition is satisfied:

$$a \leq_L b \quad \text{iff} \quad \forall (c\colon D_n \nrightarrow \emptyset)\colon ((a;c) \in L \implies (b;c) \in L).$$

Based on $\leq_L$ we can define the *Myhill-Nerode equivalence* $\equiv_L$ on cospans $a, b\colon \emptyset \nrightarrow D_n$ as follows:

$$a \equiv_L b \quad \text{iff} \quad a \leq_L b \text{ and } a \geq_L b$$

The Myhill-Nerode equivalence is called *locally finite*, if for every cospans $a\colon \emptyset \nrightarrow D_n$ the equivalence class of a is a finite set.

One can prove that the Myhill-Nerode quasi order is in fact a quasi order on $Cospan(\mathcal{HG}raph)$. It also possesses two other properties which will be important in the following. (Note that all proofs can be found in the appendix.)

Proposition 1 *Let L be a graph language over $Cospan(\mathcal{HG}raph)$. The Myhill-Nerode quasi order (relative to L) is right-monotone and the language L is upward-closed with respect to $\leq_L$.*

This proposition is the key to invariant checking. We say that a graph language L is an invariant for a rule ρ if $G \in L$ and $G \Rightarrow_\rho H$ always implies $H \in L$.

Imagine a rule ρ is given by a pair of cospans $\ell, r\colon \emptyset \nrightarrow I$ and it holds that $\ell \leq_L r$. If G is rewritten to H via ρ we have that $[G] = \ell; c$ and $[H] = r; c$ for some cospan $c\colon I \nrightarrow \emptyset$. Now $\ell \leq_L r$ implies $[G] \leq_L [H]$ (right-monotonicity) and if G is contained in L, then H is contained in L as well (upward-closure). Hence L is an invariant w.r.t. ρ. Furthermore if $\ell \not\leq_L r$, there is a cospan c violating the condition of Definition 3 and L is no invariant w.r.t. ρ. Hence we have that L is an invariant for ρ if and only if $\ell \leq_L r$.

Similar to the case of word languages we can characterize the recognizable graph languages in terms of congruence classes as shown in [BK08b]. Furthermore Ehrenfeucht et al. [EHR83] give a generalization of the Theorem of Myhill-Nerode by characterizing regular languages in terms of well quasi orders instead of equivalence classes of finite index. As an important result we can lift this theorem to the case of recognizable graph languages.

Theorem 1 (Generalized Myhill-Nerode Theorem) *Let a graph language L over $Cospan(\mathcal{HG}raph)$ be given. The following statements are equivalent:*

(i) *L is a recognizable graph language,*

(ii) *$\equiv_L$ is locally finite and L is the union of (finitely many) equivalence classes of $\equiv_L$.*

(iii) *L is upward closed with respect to some right-monotone well quasi order $\sqsubseteq_L$.*

(iv) *The Myhill-Nerode quasi order $\leq_L$ is a well quasi order.*

4 Atomic Cospans

In this section we introduce atomic graph operations which play the role of letters in the case of words. These atomic graph operations are based on the algebra of graphs originally described by Courcelle [BC87]. Each atomic graph operation is given by an atomic cospan, so that applying the graph operation to a cospan (a graph with interfaces) amounts to composing the cospan with the

atomic cospan of the operation. In the following, we will distinguish between graph operations and atomic cospans used to define them.

We assume that the set of nodes of each discrete graph D_n is $V_{D_n} = \{v_0, \dots v_{n-1}\}$. We set $\mathbb{N}_n = \{0, \dots, n-1\}$ and we denote the *disjoint union of two graphs* G_1 *and* G_2 by $G_1 \oplus G_2$. We assume that G_1 and G_2 are disjoint. Furthermore we define the *disjoint union* $f \oplus g \colon G_1 \oplus G_2 \to H_1 \oplus H_2$ *of two graph morphisms* $f \colon G_1 \to H_1$ *and* $g \colon G_2 \to H_2$ where H_1 and H_2 are disjoint as follows:

$$(f \oplus g)(v) = \begin{cases} f(v), & \text{if } v \in V_{G_1} \\ g(v), & \text{if } v \in V_{G_2} \end{cases} \quad \text{and} \quad (f \oplus g)(e) = \begin{cases} f(e), & \text{if } e \in E_{G_1} \\ g(e), & \text{if } e \in E_{G_2} \end{cases}.$$

Definition 4 (Atomic graph operations) *Restriction of the outer interface:* Let $\rho \colon D_{n-1} \to D_n$ with $\rho(v_i) = v_i$ be an arrow between two discrete graphs. We define the cospan res_n as follows: $\mathsf{res}_n \colon D_n \xrightarrow{\mathsf{id}_{D_n}} D_n \xleftarrow{\rho} D_{n-1}$.

Permutation of the outer interface: Let a permutation $\pi \colon \mathbb{N}_n \to \mathbb{N}_n$ with $\pi(i) = i+1$ for $0 \leq i < n-1$ and $\pi(n-1) = 0$ and an arrow $\sigma \colon D_n \to D_n$ with $v_i \mapsto v_{\pi(i)}$ between two discrete graphs be given. We define the cospan perm_n as follows: $\mathsf{perm}_n \colon D_n \xrightarrow{\mathsf{id}_{D_n}} D_n \xleftarrow{\sigma} D_n$.

Transposition of the outer interface: Let a transposition $\tau \colon \mathbb{N}_n \to \mathbb{N}_n$ with $\tau(0) = 1$, $\tau(1) = 0$ and $\tau(i) = i$ for $2 \leq i \leq n-1$ and an arrow $\sigma \colon V_n \to V_n$ with $v_i \mapsto v_{\tau(i)}$ between two discrete graphs be given. We define the cospan trans_n as follows: $\mathsf{trans}_n \colon D_n \xrightarrow{\mathsf{id}_{D_n}} D_n \xleftarrow{\sigma} D_n$.

Fusion of two nodes of the outer interface: Let $n > 1$ and an equivalence relation $\theta = \mathsf{id}_{V_n} \cup \{(v_0, v_1), (v_1, v_0)\}$, an arrow θ_{map} which maps every node of D_n to its θ-equivalence class, and an arrow $\varphi \colon D_{n-1} \to D$ with $v_i \mapsto [\![v_{i+1}]\!]_\theta$, where D is the discrete graph with node set $\{[\![v]\!]_\theta \mid v \in V_n\}$, be given. We define the cospan fuse_n as follows: $\mathsf{fuse}_n \colon D_n \xrightarrow{\theta_{map}} D \xleftarrow{\varphi} D_{n-1}$.

Connection of a single hyperedge: Let an edge label $A \in \Sigma$, $m \in \mathbb{N}$ with $0 \leq m \leq n$ and a hypergraph H which consists of a single hyperedge h with arity m and labeled with A be given. We define the cospan $\mathsf{connect}_n^{A,m}$ as follows: $\mathsf{connect}_n^{A,m} \colon D_n \xrightarrow{e} H \oplus D_{n-m} \xleftarrow{e} D_n$ with $e(v_i) = \mathsf{att}_i(h)$ for $0 \leq i < m$ and $e(v_i) = v_{i-m}$ otherwise.

Disjoint union with a single node: We define the cospan vertex_n as follows: $\mathsf{vertex}_n \colon D_n \xrightarrow{d^L} D_{n+1} \xleftarrow{\mathsf{id}_{D_{n+1}}} D_{n+1}$ with $d^L = \mathsf{id}_{D_n} \oplus i$ and $i \colon \emptyset \to D_1$.

The intuitions behind these atomic graph operations are as follows (see Figure 1): With the cospan res_n we can hide the last node of the outer interface of a precomposed cospan. The cospan fuse_n glues the first two nodes of the outer interface of a precomposed cospan and afterward restricts the second node of this outer interface.

The cospans trans_n and perm_n permute the outer interface of a precomposed cospan. The former maps the nodes of the outer interface in such a way that only the first two nodes are transposed. The latter permutes the nodes of the outer interface such that every node is mapped to its successor node.

In order to be able to construct new graphs the cospans vertex_n and $\mathsf{connect}_n^{A,m}$ can be used to generate new nodes and edges. By composing vertex_n with an arbitrary cospan $c \colon \emptyset \to G \leftarrow D_n$

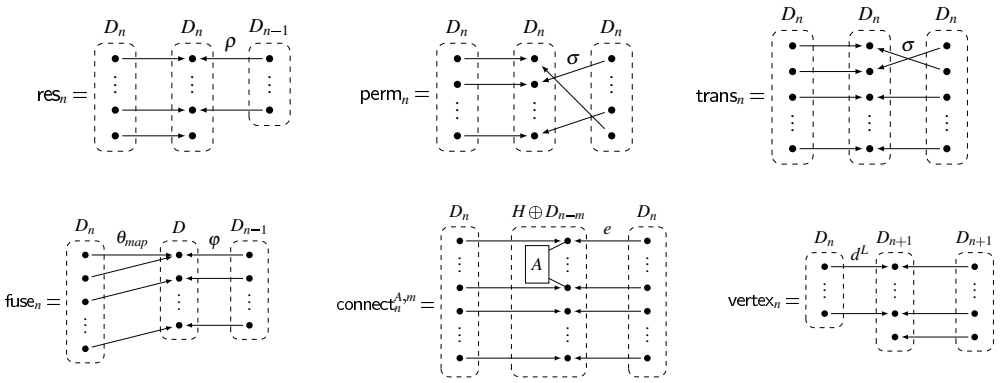


Figure 1: Graph operations

we add a single, isolated node to G and extend the outer interface of c to D_{n+1}, such that the last node of the extended outer interface is mapped to the new node. The cospan $\mathsf{connect}_n^{A,m}$ adds an A-labeled hyperedge with arity m in such a way to G that the first m nodes of the outer interface are mapped to the m nodes of the hyperedge h.

We can restrict our attention to these atomic graph operations, because any graph G (seen as a cospan of the form $\emptyset \to G \leftarrow \emptyset$) can be constructed by composing a finite number of them as shown by the next proposition.

Proposition 2 *Every cospan of the form* $c\colon D_m \xrightarrow{\varphi^L} G \xleftarrow{\varphi^R} D_n$ *where the right leg* φ^R *is injective can be constructed by a sequence* $\mathrm{op}_1,\ldots,\mathrm{op}_k$ *of atomic graph operations, i.e. c can be obtained as the composition* $c = \mathrm{op}_1;\ldots;\mathrm{op}_k$.

5 A Decidable Variant

In this section we develop an algorithm – based on the Myhill-Nerode quasi order – for checking invariants for recognizable graph languages. The algorithm takes as input an automaton functor which accepts the given graph language. In general this automaton functor has infinitely many states, since for every interface D_n ($n \in \mathbb{N}$) there exists a set of states. But for practical purposes we need an automaton functor which is finite, i.e. has only a finite number of states.

In order to get automaton functors with a finite number of state sets, we only take cospans with a bounded interface size into account.

Definition 5 (Bounded cospan) A cospan $c\colon S \nrightarrow T$ is called *bounded (by k)*, if there exist graph operations $\mathrm{op}_1,\ldots,\mathrm{op}_j$ such that $c = \mathrm{op}_1;\ldots;\mathrm{op}_j$ and for every graph operation $\mathrm{op}_i\colon D_{n_i} \nrightarrow D_{m_i}$ for $1 \leq i \leq j$ it holds that $n_i, m_i \leq k$.

Definition 6 (Bounded Myhill-Nerode quasi order) Let a natural number $k \in \mathbb{N}$ and a graph language L over $Cospan(\mathcal{HG}raph)$ be given. The quasi order $\leq_L^k$ on $Cospan(\mathcal{HG}raph)$ is called *bounded Myhill-Nerode quasi order (relative to L)*, if for arbitrary k-bounded cospans $a, b\colon \emptyset \nrightarrow D_n$

the following condition is satisfied:

$$a \leq_L^k b \quad \text{iff} \quad \forall (c\colon D_n \nrightarrow \emptyset,\ c\ k\text{-bounded})\colon ((a;c) \in L \implies (b;c) \in L).$$

The bounded Myhill-Nerode quasi order defined above gives us an over-approximation of $\leq_L$, i.e., two cospans with $a \leq_L b$ are for sure related by the relation $\leq_L^k$, but not necessarily vice versa.

Note that graphs with edges of arity more than k can not be constructed by cospans that are bounded by k. Also for edges with smaller arity it is not guaranteed that they are constructible. For example a k-grid consisting of binary edges needs interfaces of size at least k.

Since all automaton functors which accept only cospans of bounded interface size have a finite representation, we are able to consider an algorithm which computes the Myhill-Nerode quasi order relative to a given deterministic automaton functor similar to the algorithm for computing the Myhill-Nerode equivalence by pairwise comparing two states with their successor states.

But for practical purposes the algorithm is not useful due to the fact, that in general the deterministic automaton functor can be exponentially larger than the equivalent non-deterministic automaton functor. Therefore we also allow non-deterministic automaton functors as input for the algorithm. However this leads to some additional changes. Since the automaton functor is non-deterministic, for a given state there exists a set of successor states instead of a unique successor state and we cannot pairwise compare two states with their (unique) successor states. In order to circumvent this difficulty, we allow an "one-sided error" by taking a stronger relation than the Myhill-Nerode quasi order. Roughly, we are under-approximating language inclusion via some form of simulation. A relation R on the states of an automaton functor $\mathscr{A}$ is a simulation, if the following condition is satisfied:

$$s_1\, R\, s_2 \implies \left(s_1 \in \mathrm{F}^{\mathscr{A}} \Rightarrow s_2 \in \mathrm{F}^{\mathscr{A}}\right) \wedge \forall \mathrm{op}\colon \forall s_1' \in \mathscr{A}(\mathrm{op})(s_1)\colon \exists s_2' \in \mathscr{A}(\mathrm{op})(s_2)\colon (s_1'\, R\, s_2').$$

A state t_2 *simulates* a state t_1, denoted by $t_1 \preceq t_2$, if $t_1\, R\, t_2$ holds for some simulation R.

Definition 7 (Bounded simulation) Let L be a graph language over $\mathit{Cospan}(\mathcal{HG}raph)$ and $\mathscr{A}$ an automaton functor, which accepts the language L. The quasi order $\leq_{\mathscr{A}}^k$ is called *bounded simulation (relative to L)*, if for arbitrary, k-bounded cospans $a, b\colon \emptyset \nrightarrow D_m$ the following condition is satisfied:

$$a \leq_{\mathscr{A}}^k b \quad \text{iff} \quad \forall s_1 \in \mathscr{A}(a)(\mathrm{I}^{\mathscr{A}})\colon \exists s_2 \in \mathscr{A}(b)(\mathrm{I}^{\mathscr{A}})\colon s_1 \preceq s_2.$$

Replacing the (bounded) Myhill-Nerode quasi order by the (bounded) simulation relation results in fact in an one-sided error, as the next proposition shows:

Proposition 3 *Let $n, k \in \mathbb{N}$ with $n \leq k$, $a, b\colon \emptyset \nrightarrow D_n$ be cospans and $\mathscr{A}$ be the automaton functor which accepts the language L. If $a \leq_{\mathscr{A}}^k b$ holds, then $a \leq_L^k b$ holds. The inverse direction holds if $\mathscr{A}$ is deterministic.*

Algorithm 1 on page 9 computes $\leq_{\mathscr{A}}^k$ as defined above. Note that this is a fixed-point algorithm computing the greatest fixed-point. The relations $\preceq^i$ (one for each interface size) first contain all possible pairs of states and are suitably refined in each step. First, we delete all pairs, where the first state is final and the second is not. Then, for all pairs still in the relation we check whether

each transition from the first state can be mimicked by the second such that the resulting states are in the relation. If no more pairs can be deleted we have reached a fixed-point and terminate. Then it is left to check whether

Algorithm 1 `CheckSimRelated`$(a,b,k,\mathscr{A})$

Input: Bounded cospans $a,b\colon \emptyset \nrightarrow D_n$ with $n \leq k$, an automaton functor $\mathscr{A}$
Output: **true**, if $a \leq^k_{\mathscr{A}} b$ and **false**, if $a \not\leq^k_{\mathscr{A}} b$

```
set ⪯^i = 𝒜(D_i) × 𝒜(D_i) for all 0 ≤ i ≤ k
for all s_0 ∈ F^𝒜, s_1 ∈ 𝒜(∅) \ F^𝒜 do
    delete (s_0, s_1) ∈ ⪯^0
repeat
    for all (s_0, s_1) ∈ ⪯^i with 0 ≤ i ≤ k do
        for all op ∈ {connect_i^{A,m}, fuse_i, perm_i res_i, trans_i, vertex_i} do
            for all s'_0 ∈ 𝒜(op)(s_0) do
                if there exists no s'_1 ∈ 𝒜'(op)(s_1), such that (s'_0, s'_1) ∈ ⪯^i then
                    delete (s_0, s_1) from ⪯^i
until no deletion has been performed in the last iteration
for all i ∈ I^𝒜 do
    for all s_0 ∈ 𝒜(a)(i) do
        if there exists no state s_1 ∈ 𝒜(b)(i), such that (s_0, s_1) ∈ ⪯^n then
            return false
return true
```

Theorem 2 *Let an automaton functor $\mathscr{A}$ and two bounded cospans $a,b\colon \emptyset \nrightarrow D_n$ with $n \leq k$ be given. Then $a \leq^k_{\mathscr{A}} b$ holds, if and only if* `CheckSimRelated`$(a,b,k,\mathscr{A})$ *returns* `true`.

We implemented the algorithm in a naive way: our implementation explicitly stores the relations $\preceq^i$ in tables and iterates until no further changes occur. More details about the run-time and memory requirement of the naive implementation are given in the next section; some ideas for significant improvement are presented as future work in the conclusion.

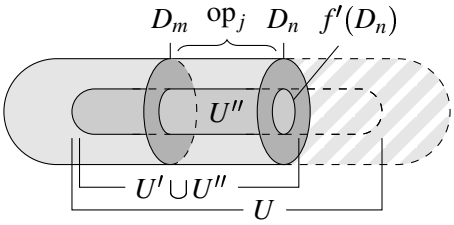
6 Short Example

In this section we consider a multi-user file system where the access to the system is controlled by several rules in order to guarantee some consistency properties. The case study was inspired by [KMP02]. As in most cases, the violation of these consistency properties can be modeled by the occurrence of one or more forbidden graphs. Therefore, we first introduce a k-bounded automaton functor $\mathscr{A}$, i.e. an automaton functor processing only k-bounded graphs, which accepts every graph $[G]$ which contains a specified subgraph U.

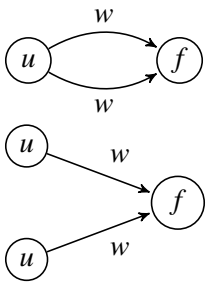
The idea behind this automaton functor is as follows: The automaton functor used in this example contains a state set $\mathscr{A}(D_i)$ for every discrete interface D_i, $0 \leq i \leq k$. Every state in each state set stores two kinds of information: on the one hand the subgraph U' of U which has already been read and on the other hand a partial function f from V_{D_i} to $V_{U'}$ describing which vertices of U' are contained in the interface D_i. By Proposition 2, we can restrict the automaton functor to

accept only atomic graph operations (see Section 4), since every cospan $[H]$ can be decomposed to a sequence of atomic graph operations $\mathrm{op}_1,\ldots,\mathrm{op}_\ell$ such that $[H] = \mathrm{op}_1;\ldots;\mathrm{op}_\ell$. For every atomic graph operation $\mathrm{op}_j\colon D_m \nrightarrow D_n$ with $1 \le j \le \ell$, $m,n \in \{0,\ldots,k\}$ containing a subgraph U'' of U and a state $(U',f) \in \mathscr{A}(D_m)$ the successor state $(U' \cup U'', f') \in \mathscr{A}(D_n)$ is computed by adding the new subgraph U'' to the subgraph U' and updating the partial function f according to op_j resulting in the partial function f' (see image below). Note that op_j might contain various subgraphs U'' and hence the automaton is heavily non-deterministic. More details concerning the construction of this automaton functor can be found in [Blu08].

We can show that we obtain a functor which guarantees that the decomposition of the cospan $[H]$ does not affect the acceptance behavior of the automaton functor. The set of start states $\mathrm{I}^{\mathscr{A}}$ contains only the state $(\emptyset,\emptyset)$ consisting of the empty graph and the empty partial function. The set of acceptance states $\mathrm{F}^{\mathscr{A}}$ contains only the state $(U,\emptyset)$ consisting of the wanted subgraph and the empty partial function.

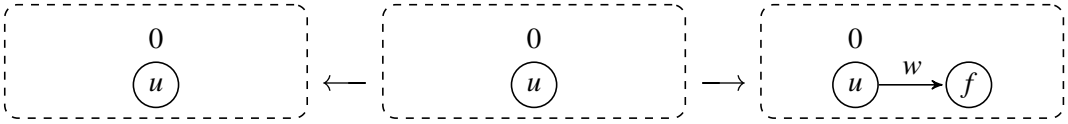


Now we want to use this automaton functor for the verification of the multi-user file system. We consider two properties which describe when the consistency of the multi-user file system is violated. The system is in a consistent state as long as these properties are *not* satisfied. The first property is the double write access of a user to a file (*double access*), i.e. a user has two times a write access to the same file at the same time. The second property is the write access of two different users to the same file at the same time (*two users*). These two properties can be modeled by the following two graphs, where nodes labeled with u (resp. f) denote users (resp. files) and edges from a user-node to a file-node labeled with w (resp. r) denote a write (resp. a read) access of that user to that file:

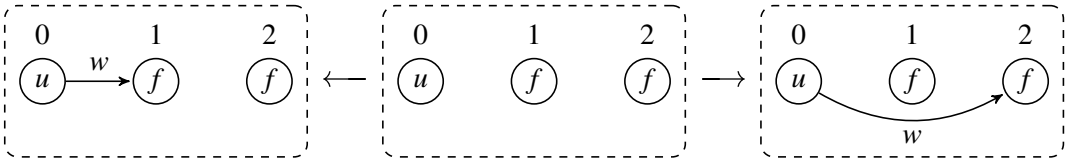


Note that it is not forbidden that a user has more than one read access to a file at the same time and that two or more users can have read access to the same file at the same time even if one user has write access to that file. Since recognizable languages are closed under boolean operations and with the considerations above we can now construct an automaton functor that recognizes all graphs violating one of the two properties, i.e., all graphs that contain either of the two subgraphs.

Furthermore, the multi-user file system offers the usual operations such as adding and removing users, creating, deleting and requesting files as well as switching, dispossessing and transferring access rights. In the following, we will show with the rules "User creates new file" and "User requests file" how these file system operations can be modeled as DPO rewrite rules. The rule "User creates new file" applied for some user u creates a new file f and gives the user a write access to this file. It can be modeled by the following span:



The rule "User requests file" applied for some user u sets the write access of this user from the current file to some other existing file. The following span models this rule:



Since every rewrite rule can be considered as two cospans ℓ and r (see Subsection 2.1) which are the left and right hand side of the corresponding rewrite rule, we can verify the consistency of this multi-user file system by checking, if the language of all graphs containing none of the forbidden subgraphs is an invariant for each rule. Since the automaton functor accepts the complement of this language, i.e., all graphs that *do* contain one of the forbidden subgraphs, we perform a backwards analysis on each rewrite rule and check whether $r \leq^k_{\mathscr{A}} \ell$. If r is related to ℓ, then the original rewrite rule does not violate the consistency of the multi-user file system. After the application of the rule the consistency of the system is violated only if it was already violated before the rule application, hence the language is verified to be an invariant.

We now use the algorithm described in the previous section to check the rewrite rules mentioned above. For all interface sizes that we checked the result of the algorithm is that the language is an invariant w.r.t. the first rule, but not w.r.t. the second rule. This is clear, since a user can request write access to a file, to which another user has already write access. Note also that, due to the under-approximation by simulations, there are actually rules which are correct, but are not recognized as such by the algorithm.

Although the example is rather small, the computed simulation relation becomes very large quickly. Table 1 presents the size of the simulation relation (according to the number of pairs contained in the relation) and the run-time of the implementation of Algorithm 1 for some interface sizes. The tests were performed on a Linux machine with a Xeon Dualcore 5150 processor and 2 GB of available main memory.

	Maximum interface size				
	0	1	2	3	4
Size (in pairs)	400	3.425	31.314	323.995	$\approx 3{,}7 \cdot 10^6$
Run-time (in seconds)	<1s	<1s	<1s	2s	26s

Table 1: Size of the simulation relation and run-time of the algorithm

Note that for interfaces with a size more than 4 the size of the simulation relation exceeds the amount of main memory. Nevertheless it is possible to verify all rewrite rules which have a interface size up to 4.

7 Conclusions

The notion of recognizable graph language used in this paper has been introduced in [BK08b] and is strongly related to [Cou90, Gri03, BK06]. Especially the notion of recognizability considered

here is equivalent to Courcelle's notion. For a detailed comparison see [BK08b]. In [BK08a] a weaker notion of graph automata is introduced.

Invariant checking for graph transformation rules has already been considered in several papers: in [FL97, BPR03] shape types and shapes are introduced in order to describe graph languages. Both papers propose algorithms that analyze each rule and check whether (and how) it may change the shape of a graph. In order to describe shapes the former uses context-free grammars whereas the latter uses more expressive graph reduction systems, that are able to express properties such as balancedness of trees. In [HPR06] a method for computing weakest preconditions of application conditions, which are equivalent to first-order graph logic, is presented. This method can also be used for invariant checking, by showing that for every rule the weakest precondition of the invariant is implied by the invariant. Note that, in general, recognizable graph languages are more expressive than first-order logic since every monadic second-order graph logic formula is known to specify a recognizable graph language [Cou90]. Another related work [BBG+06] considers graph patterns consisting of negative and positive components and shows that they are invariants via an exhaustive search. Interestingly, this method made efficient by a symbolic algorithm based on binary decision diagrams, an idea that we are trying to reuse in a somewhat different setting (see remarks below).

We have not yet compared the effectiveness of our approach to these other approaches in detail, but our method is different from all the others in that it is based on the Myhill-Nerode quasi order.

Our approach suffers from the restriction that we have to work with k-bounded cospans. Especially we first over-approximate the relation $\leq_L$ by $\leq_L^k$ (by introducing k-boundedness), which is subsequently under-approximated by $\leq_{\mathscr{A}}^k$ (by using simulation instead of language inclusion). While it is difficult to imagine how to avoid the restriction to interfaces up to size k, the determinization of the automaton functor $\mathscr{A}$, which would avoid the under-approximation, should be achievable if we use a more succinct representation of automaton functors. We are currently experimenting with the representation of automaton functors (which are basically very large relations) with binary decision diagrams (BDDs), which are well-suited for the compact representation of large (but finite) relations. Our experiments have so far been very promising. With BDDs we can handle much larger interfaces and we expect to obtain less memory usage and better run-times.

Finally, decomposing a graph into atomic cospans is basically equivalent to the path decomposition of a graph and checking whether a graph is contained in the language is hence linear-time for graphs of bounded pathwidth. For efficiency reasons it would be more suitable to consider generalizations of tree automata that can handle tree decompositions of graphs, as it is similarly done in the work by Courcelle. Hence we are currently investigating tree automata and their generalization to graphs.

Bibliography

[BBG+06] B. Becker, D. Beyer, H. Giese, F. Klein, D. Schilling. Symbolic invariant verification for systems with dynamic structural adaptation. In *Proc. of ICSE '06 (International Conference on Software Engineering)*. Pp. 72–81. ACM, 2006.

[BC87] M. Bauderon, B. Courcelle. Graph Expressions and Graph Rewritings. *Mathematical Systems Theory* 20(2-3):83–127, 1987.

[BK06] S. Bozapalidis, A. Kalampakas. Recognizability of graph and pattern languages. *Acta Informatica* 42(8/9):553–581, 2006.

[BK08a] S. Bozapalidis, A. Kalampakas. Graph automata. *Theoretical Computer Science* 393:147–165, 2008.

[BK08b] H. J. S. Bruggink, B. König. On the Recognizability of Arrow and Graph Languages. In *Proc. of ICGT '08*. Springer, 2008. LNCS 5214.

[Blu08] C. Blume. Graphsprachen für die Spezifikation von Invarianten bei verteilten und dynamischen Systemen. Master's thesis, Universität Duisburg-Essen, November 2008. (in German).

[BPR03] A. Bakewell, D. Plump, C. Runciman. Checking the Shape Safety of Pointer Manipulations. In *Proc. of RelMiCS '03*. Pp. 48–61. 2003.

[Cou90] B. Courcelle. The Monadic Second-Order Logic of Graphs. I. Recognizable Sets of Finite Graphs. *Inf. Comput.* 85(1):12–75, 1990.

[EHR83] A. Ehrenfeucht, D. Haussler, G. Rozenberg. On Regularity of Context-Free Languages. *Theor. Comput. Sci.* 27:311–332, 1983.

[FL97] P. Fradet, D. Le Métayer. Shape Types. In *Proc. of POPL '97*. Pp. 27–39. ACM, 1997.

[GMM94] P. H. B. Gardiner, C. E. Martin, O. de Moor. An algebraic construction of predicate transformers. *Sci. Comput. Program.* 22(1-2):21–44, 1994.

[Gri03] G. Griffing. Composition-representative subsets. *Theory and Applications of Categories* 11(19):420–437, 2003.

[HPR06] A. Habel, K.-H. Pennemann, A. Rensink. Weakest Preconditions for High-Level Programs. In *Proc. of ICGT '06*. Springer, 2006. LNCS 4178.

[Kel82] G. M. Kelly. *Basic Concepts of Enriched Category Theory*. Cambridge University Press, 1982.

[KMP02] M. Koch, L. V. Mancini, F. Parisi-Presicce. Decidability of Safety in Graph-based Models for Access Control. In *Proceedings of the 7th European Symposium on Research in Computer Security*. Pp. 229–243. Springer, 2002. LNCS 2502.

[LV94] A. de Luca, S. Varricchio. Well Quasi-Orders and Regular Languages. *Acta Inf.* 31(6):539–557, 1994.

[SS05] V. Sassone, P. Sobociński. Reactive Systems over Cospans. In *LICS*. Pp. 311–320. 2005.

Electronic Communications of the EASST
Volume of the Pre-proceedings of GT-VMT (2010)

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

Efficient Analysis of Permutation Equivalence of Graph Derivations Based on Petri Nets

Frank Hermann , Andrea Corradini , Hartmut Ehrig , Barbara König

14 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

Efficient Analysis of Permutation Equivalence of Graph Derivations Based on Petri Nets

Frank Hermann [1], **Andrea Corradini** [2], **Hartmut Ehrig** [1], **Barbara König** [3]

[1] [frank,ehrig]@cs.tu-berlin.de, Institut für Softwaretechnik und Theoretische Informatik, Technische Universität Berlin, Germany
[2] andrea(at)di.unipi.it, Dipartimento di Informatica, Università di Pisa, Italy
[3] barbara_koenig(at)uni-due.de, Abteilung für Informatik und Angewandte Kognitionswissenschaft, Universität Duisburg-Essen, Germany

Abstract: In the framework of graph transformation systems with Negative Application Conditions (NACs) the classical notion of switch equivalence of derivations is extended to *permutation equivalence*, because there are intuitively equivalent derivations which are not switch-equivalent if NACs are considered. By definition, two derivations are permutation-equivalent, if they respect the NACs and disregarding the NACs they are switch equivalent. A direct analysis of permutation equivalence is very complex in general, thus we propose a much more efficient analysis technique. For this purpose, we construct a Place/Transition Petri net, called dependency net, which encodes the dependencies among rule applications of the derivation, including the inhibiting effects of the NACs.

The analysis of permutation equivalence is important for analysing simulation runs within development environments for systems modelled by graph transformation. The application of the technique is demonstrated by a graph transformation system within the context of workflow modelling. We show the effectiveness of the approach by comparing the minimal costs of a direct analysis with the costs of the efficient analysis applied to a derivation of our example system.

Keywords: graph transformation, Petri nets, process analysis, adhesive categories

1 Introduction

Given a workflow of a system, it is often interesting to know whether the workflow can be improved, by executing the tasks in a different order, which might be more convenient for the user or preferable from an efficiency point of view. If the workflow is modelled by a Petri net, representing a deterministic process, these questions can be fairly easily answered: processes incorporate a notion of concurrency that can be exploited to rearrange the tasks, while still respecting causality.

In this paper we consider workflow models with two further dimensions, which considerably complicate the problem: first, we work in the general setting of (weak) adhesive categories [LS05, EEPT06] where we can model systems with an evolving topology, such as (attributed) graph transformation systems, in contrast to systems with a static structure. For the sake of con-

ciseness, the definitions and results in this paper are presented for *graph* transformation systems, and we refer to the companion technical report [HCEK10] for the general notions based on adhesive categories. As a second dimension, we take into account Negative Application Conditions (NACs) that are used to ensure the "absence" of forbidden structures when executing a transformation step: NACs significantly improve the specification formalisms based on transformation rules leading to more compact and concise models as well as increased usability and as a matter of fact they are widely used in non-trivial applications. The presence of NACs leads to more complex interdependencies of tasks.

For this reason, we introduce a notion of permutation equivalence on derivations with NACs, which is coarser and more adequate than the switch equivalence in the double-pushout (DPO) approach including NACs. As defined in [Her09] two derivations are called permutation-equivalent, if they respect the NACs and disregarding the NACs they are switch-equivalent. Using the notion of switch equivalence with NACs directly does not lead to all permutation-equivalent derivations of a given derivation in general. The main remaining problem is how to derive the complete set of all permutation-equivalent derivations to a given one. For this purpose, we construct a subobject transformation system (STS) via a standard colimit construction and from this STS we construct a dependency net, given by a standard P/T Petri net, which includes a complete account of the inhibiting effects of the NACs. The main result shows that complete firing sequences of this net are one-to-one with derivations that are permutation-equivalent to the given derivation, allowing us to derive the complete set of permutation-equivalent derivations. Finally, for a given derivation of a simple example system with NACs, we perform a detailed complexity analysis of the cost of identifying all permutation equivalent derivations using the reduction to a Petri net and its reachability graph, and compare it with a lower bound of the costs for a direct analysis, i.e. for computing all shift-equivalent derivations first, and then filtering out the ones which do not respect the NACs. We obtain a significant improvement in speed, which shows that the proposed technique can be efficient for many applications which involve the generation of permutation-equivalent derivations. Furthermore, the constructed P/T Petri net can be used to derive specific permutations without generating the complete set first. In the context of workflow analysis, both goals are of central interest for the modelling of a system.

The structure of the paper is as follows. Sec. 2 reviews the main concepts of permutation equivalence for graph transformation systems. The construction of the dependency net is presented in Sec. 3 and in our main result it is shown to be sound and complete for computing the set of permutation-equivalent derivations. Sec. 4 validates the efficiency of the analysis based on an extended version of the running example. Finally, Sec. 5 sums up the main results, discusses related work, and points out aspects of future work.

2 Permutation Equivalence

In this section we review the standard switch equivalence and the recently introduced permutation equivalence [Her09] for graph transformation systems based on the double pushout (DPO) approach. The running example of this paper illustrates that there are derivations which are intuitively equivalent and also permutation-equivalent but not switch-equivalent.

Definition 1 (Graph Transformation System with NACs) A *rule* $p = (L_p \xleftarrow{l} K_p \xrightarrow{r} R_p)$ is a pair of injective graph morphisms. A *Negative Application Condition (NAC)* for a rule p is an injective graph morphism $n : L_p \hookrightarrow N$, having the left-hand side of p as source. A *rule with NACs* is a pair $\langle p, \mathbf{N} \rangle$ where p is a rule and $\mathbf{N} = \{n_i : L_p \hookrightarrow N_i\}_{i \in I}$ is a finite set of NACs for p. A *match* of a rule p in a graph G is an injective graph morphism[1] $m : L_p \hookrightarrow G$; match m *satisfies the NAC* $n : L_p \hookrightarrow N$ *for* p, written $m \models n$, if there is no arrow $g : N \to G$ such that $g \circ n = m$.[2] We say that there is a *direct derivation* $G \xRightarrow{p,m} H$ from an object G to H using a rule with NACs $\langle p, \mathbf{N} \rangle$ and a match $m : L_p \to G$, if there are two pushouts (1) and (2) in **Graphs**, as depicted. A derivation *respects the NACs*, if $m \models n$ for each NAC $(n : L_p \hookrightarrow N) \in \mathbf{N}$. A typed *graph transformation system* (*GTS*) with NACs is a tuple $\mathcal{G} = \langle Q, \pi_N \rangle$ where Q is a set of rule names, and π_N maps each name $q \in Q$ to a rule with NACs $\pi_N(q) = \langle \pi(q), \mathbf{N}_q \rangle$ in the category $\mathbf{Graphs}_{TG}$ of graphs typed over a given type graph *TG*. A *derivation (respecting NACs)* of $\mathcal{G}$ is a sequence $G_0 \xRightarrow{q_1,m_1} G_1 \cdots \xRightarrow{q_n,m_n} G_n$, where $q_1, \ldots, q_n \in Q$ and $d_i = G_{i-1} \xRightarrow{\pi(q_i),m_i} G_i$ are direct derivations (respecting NACs) for $i \in 1, \ldots, n$. Sometimes we denote a derivation as a sequence $d = d_1; \ldots; d_n$ of direct derivations.

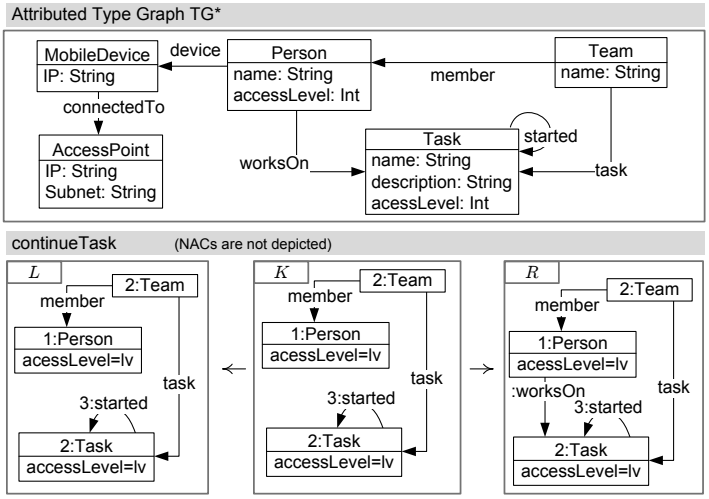


Figure 1: Part of attributed transformation system GS^*, modeling mobile adhoc networks

Example 1 (Graph Transformation System with NACs) *Fig. 1 shows a part of an attributed graph grammar for modelling a workflow system in mobile adhoc networks, where persons can be assigned to teams and tasks and they can change their location implying that their mobile communication devices may need to reconnect to new access points. In order to simplify the further constructions we will use the reduced version of this grammar in Fig. 2. The* type graph *TG shows that nodes in the system represent either persons or tasks: a task is active if it has a*

[1] In the general case NAC-morphisms $(n : L \to N)$ and matches are not required to be injective. For the general case, our technique can be extended by the results in [HE08].

[2] Intuitively, the image of L_p in G cannot be extended to an image of the "forbidden context" N.

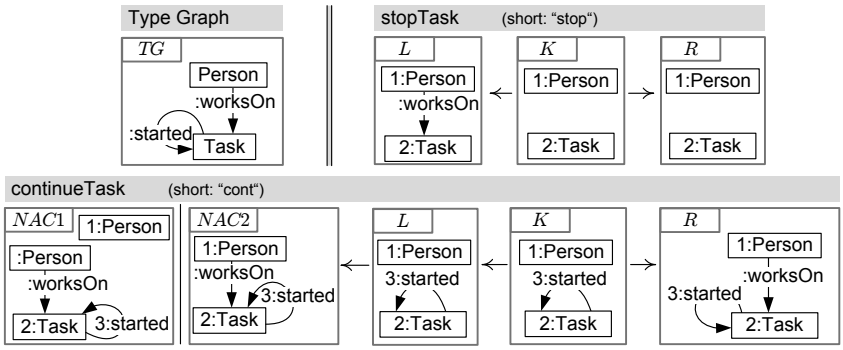


Figure 2: Reduced transformation system *GS* as running example

":started" loop, and it can be assigned to a person with a ":worksOn" edge. Rule "stopTask" cancels the assignment of a task to a person; rule "continueTask" instead assigns the task, and it has two NACs to ensure that the task is not assigned to a person already. Fig. 3 shows two derivations respecting NACs of GS. In derivation d the only task is first continued by "1:Person", and then, after being stopped, by "2:Person". In d′ the roles of the two Persons are inverted.

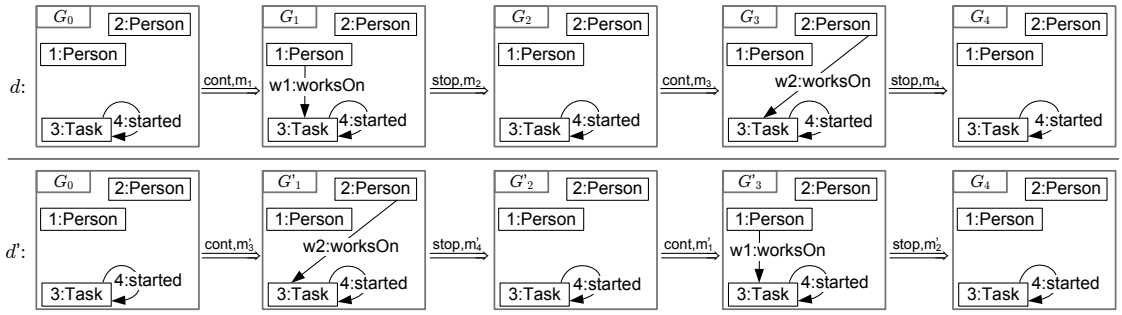
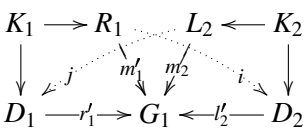


Figure 3: Derivation d (respecting NACs) of *GS* and permutation-equivalent derivation d'

The classical theory of the DPO approach (without NACs) introduces an equivalence among derivations which relates derivations that differ only in the order in which independent direct derivations are performed (see [Kre86, BCH+06]). The *switch equivalence* is based on the notion of *sequential independence* and on the *Local Church-Rosser theorem*. This is briefly summarised in the next definition.

Definition 2 (Switch Equivalence on Derivations) Let $d_1 = G_0 \xRightarrow{p_1,m_1} G_1$ and $d_2 = G_1 \xRightarrow{p_2,m_2} G_2$ be two direct derivations. Then they are *sequentially independent* if there exist arrows $i : R_1 \to D_2$ and $j : L_2 \to D_1$ such that $l'_2 \circ i = m'_1$ and $r'_1 \circ j = m_2$ (see the diagram on the right, which shows part of the derivation diagrams). If d_1 and d_2 are sequentially independent, then according to the Local Church Rosser Theorem (Thm. 5.12 in [EEPT06]) they can be "switched" obtaining direct derivations $d'_2 = G_0 \xRightarrow{p_2,m_2} G'_1$ and $d'_1 = G'_1 \xRightarrow{p_1,m_1} G_2$, which apply the two rules in the opposite order.



Now, let $d = (d_1; \ldots; d_k; d_{k+1}; \ldots; d_n)$ be a derivation, where d_k and d_{k+1} are two sequentially independent direct derivations, and let d' be obtained from d by switching them according to

the Local Church Rosser Theorem. Then, d' is a *switching of* d, written $d \overset{sw}{\sim} d'$. The *switch equivalence*, denoted $\overset{sw}{\approx}$, is the smallest equivalence on derivations containing both $\overset{sw}{\sim}$ and the relation $\cong$ for isomorphic derivations.

Corresponding notions of parallel and sequential independence have been proposed for graph transformation systems with NACs [HHT96, LEO06]. However, the derived notion of switch equivalence does not identify all intuitively equivalent derivations. The reason is that, in presence of NACs, there might be an equivalent permutation of the direct derivations that cannot be derived by switch equivalence. Looking at d in Fig. 3 there is no pair of consecutive direct derivations which is sequentially independent if NACs are considered. However, the derivation d' should be considered as equivalent. There are also examples in which even the switching of blocks of several steps would not lead to all permutation-equivalent derivations. This brings us to the following, quite natural notion of permutation equivalence of derivations respecting NACs, first proposed in [Her09]. Note that for permutation-equivalent derivations $d \overset{\pi}{\approx} d'$ the sequence of rules used in d' is a permutation of those used in d.

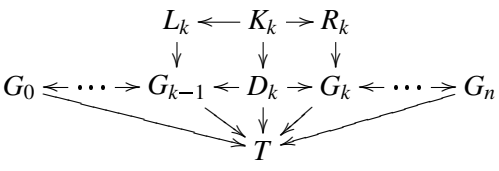
Definition 3 (Permutation Equivalence of Derivations) Two derivations d and d' respecting NACs are *permutation equivalent*, written $d \overset{\pi}{\approx} d'$ if, disregarding the NACs, they are switch equivalent as for Def. 2.

3 Dependency Net of a Derivation

In order to efficiently analyse permutation equivalence of derivations we introduce the construction of the dependency net of a given derivation with NACs. This Place/Transition Petri net purely encodes the dependencies between the derivation steps. The reachability graph of this net with initial marking determines the class of derivations which are permutation-equivalent to a given one.

The construction of the dependency net is based on the construction of the subobject transformation sytem (STS) of a given derivation d according to [CHS08] and its extension to NACs in [Her09]. Subobjects of a graph G form the category of subobjects $\mathbf{Sub}(G)$, which contains subgraphs of G[3] as objects and injective graph morphisms $m : G_1 \to G_2$ between subgraphs as morphisms, where m is required to respect the injective embeddings of G_1 and G_2 to G. We will write $G_1 \cap G_2$ for the componentwise intersection and $G_1 \cup G_2$ for the componentwise union of subgraphs of G, where items are identified with respect to the injective embeddings of G_1 and G_2 into G.

In order to construct the STS for a derivation $d = (d_1;\ldots;d_n)$ we compute the colimit T of the sequence of DPO diagrams, where all morphisms are injective. Thus, all objects and morphisms of this diagram are in the category $\mathbf{Sub}(T)$. The NACs of the rules do not occur in this diagram.



[3] More formally, a subgraph is given by an equivalence class of injective graph morphisms to G, such that the image of all morphism is equal.

Definition 4 (STS of a derivation) A *subobject transformation system* $\mathcal{S} = \langle T, Q, \pi \rangle$ consists of a super object T, a set of rule names Q, and a function π, which maps a name $q \in Q$ to a *rule*, i.e., to a triple $\pi(q) = \langle L_q, K_q, R_q \rangle$ of subobjects of T such that $K_q = L_q \cap R_q$.

Now, let $\mathcal{G} = \langle Q, \pi \rangle$ be a graph transformation system, and let $d = (G_0 \overset{q_1,m_1}{\Longrightarrow} \ldots \overset{q_n,m_n}{\Longrightarrow} G_n)$ be a derivation of $\mathcal{G}$. The *STS generated from* d is defined as $STS(d) = \langle T, P, \hat{\pi} \rangle$, where T is the colimit object of the diagram underlying the derivation d, $P = \{k \mid d_k = (G_{k-1} \overset{q_k,m_k}{\Longrightarrow} G_k)$ is a step of $d\}$, and $\hat{\pi}(k) = \langle L_k, K_k, R_k \rangle$, where $q_k = (L_k \leftarrow K_k \rightarrow R_k)$.

For the rest of the paper, we consider only derivations such that the colimit T is a *finite object*, i.e. $\mathbf{Sub}(T)$ is a finite lattice. This is guaranteed if each rule of $\mathcal{G}$ has finite left- and right-hand sides, and if the start object of the derivation is finite. The generation of an STS with NACs from a given derivation works as in Definition 4, but additionally each rule will be equipped with a list of NACs, i.e., those obtained as "instances" of the original NACs in the colimit object T. Note that one original NAC can have several instances, but also not a single one.

Definition 5 (Instantiated NACs) Let $d = d_1; \ldots; d_k; \ldots; d_n$ be a derivation respecting NACs and let T be the colimit object of the derivation. Let $\langle p, \mathbf{N} \rangle$ be the rule with NACs used in direct derivation d_k and let $NACS(p) = \{n : L_p \hookrightarrow N \mid n \in \mathbf{N}\}$. The set of all instantiated NACs in T of the NACs of a rule p is given by $NACS_T(p) = \{N \overset{t_N}{\rightarrow} T \text{ in } \mathbf{Sub}(T) \mid n \in \mathbf{N}, \text{ s.t. } t_N \circ n = t_L\}$ for $L_p \overset{t_L}{\rightarrow} T$ in $\mathbf{Sub}(T)$.

Definition 6 (STS of a Derivation with NACs) Let $\mathcal{G}$ be a GTS with NACs and let d be a derivation of $\mathcal{G}$ respecting NACs. The *STS with NACs generated by* d is given by $STS_N(d) = \langle T, P, \hat{\pi}_N \rangle$, where T and P are as in Def. 4, $\hat{\pi}_N(k) = \langle \hat{\pi}(k), \mathbf{N}_k \rangle$, $\hat{\pi}(k)$ is as in Def. 4, and $\mathbf{N}_k$ is an arbitrary but fixed linearisation of the instantiated NACs $NACS_T(p_k)$ as in Def. 5, where p_k is the rule of $\mathcal{G}$ used in d_k.

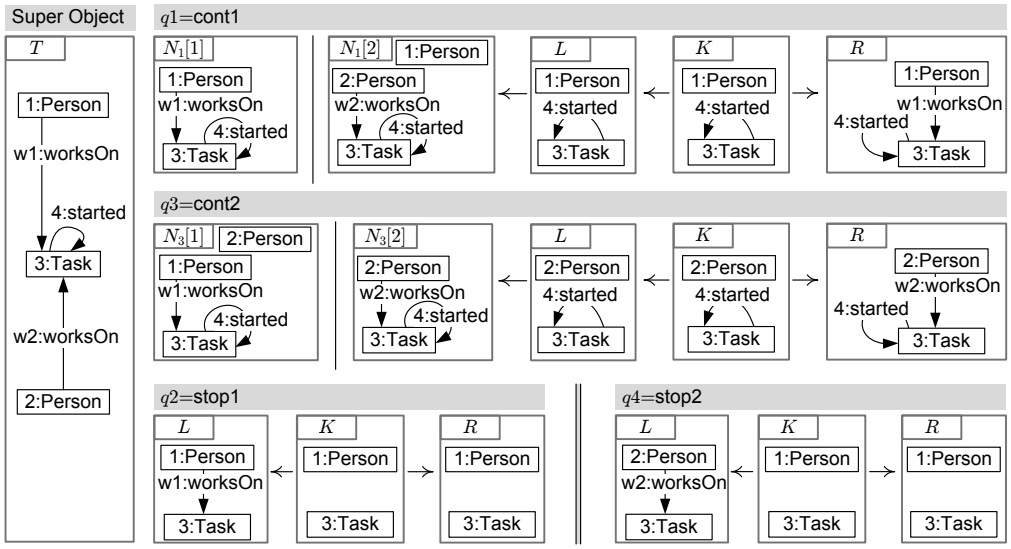


Figure 4: Derived Subobject Transformation System $STS_N(d)$

Example 2 (Derived STS $STS_N(d)$) *For the derivation d in Ex. 1 we derive the STS as shown in Fig. 4. The super object T is derived by taking the first graph of the derivation and adding the items, which are created during the transformation, i.e. the two edges of type "worksOn". The derivation d involves the rules "continueTask" and "stopTask" and thus, the derived STS contains the rule occurrences "1 $\triangleq$ cont1", "2 $\triangleq$ stop1", "3 $\triangleq$ cont2" and "4 $\triangleq$ stop2", where the NACs of the rule "continueTask" are instantiated.*

The following relations between the rules of an STS with NACs specify the possible dependencies among them: the first four relations are discussed in [CHS08], while the last two are introduced in [Her09]. In our case the STS with NACs is generated from a derivation d according to Def. 6.

Definition 7 (Relations on Rules) Let q_1 and q_2 be two rules in an STS with NACs $\mathcal{S} = (T,P,\pi_N)$ with $\pi_N(q_j) = (\langle L_j,K_j,R_j\rangle,\mathbf{N}_j)$ for $j \in \{1,2\}$ and $\mathbf{N}_j = (N_j[i])_{i=1..n_j}$. The relations on rules are defined on P as follows:

Name	Notation	Condition
Read Causality	$q_1 <_{rc} q_2$	$R_1 \cap K_2 \not\subseteq K_1$
Write Causality	$q_1 <_{wc} q_2$	$R_1 \cap L_2 \not\subseteq K_1 \cup K_2$
Deactivation	$q_1 <_{d} q_2$	$K_1 \cap L_2 \not\subseteq K_2$
Independence	$q_1 \Diamond q_2$	$(L_1 \cup R_1) \cap (L_2 \cup R_2) \subseteq K_1 \cap K_2$
Weak NAC Enabling	$q_1 <_{wen[i]} q_2$	$1 \leq i \leq \lvert\mathbf{N}_2\rvert \wedge L_1 \cap N_2[i] \not\subseteq K_1 \cup L_2$
Weak NAC Disabling	$q_1 <_{wdn[i]} q_2$	$1 \leq i \leq \lvert\mathbf{N}_1\rvert \wedge N_1[i] \cap R_2 \not\subseteq L_1 \cup K_2$

Read causality specifies that rule q_1 produces an item that is read by q_2, but not deleted by q_2 and in the case of write causality we have that q_2 also deletes such an item. Deactivation occurs when rule q_2 deletes an item that is read by q_1, but not created and two rule occurrences are independent if they overlap only on items that are neither produced nor deleted by one of the rules. Rule q_1 weakly enables the rule q_2 at i if q_1 deletes a forbidden part q_2, i.e. an item of the i-th NAC of q_2 that is not contained in L_2. The rule q_2 weakly disables q_1 at i if q_2 produces a piece of the i-th NAC of q_1. It is worth stressing that the relations introduced above are not transitive in general.

Example 3 (Relations on Rules) *The rules of $STS_N(d)$ in Fig. 4 are related by the following dependencies. For write causality we have "cont1 $<_{wc}$ stop1" and "cont2 $<_{wc}$ stop2". Weak enabling/disabling are shown in the table below, while read causality and deactivation are empty.*

Weak Enabling		*Weak Disabling*	
$stop1 <_{wen[1]} cont1$	$stop2 <_{wen[2]} cont1$	$cont1 <_{wdn[1]} cont1$	$cont2 <_{wdn[2]} cont2$
$stop1 <_{wen[1]} cont2$	$stop2 <_{wen[2]} cont2$	$cont2 <_{wdn[1]} cont1$	$cont1 <_{wdn[2]} cont2$

Based on the STS of a derivation, we now present the construction of its "dependency net",

given by a P/T Petri net which specifies only the dependencies between the derivation steps. All details about the internal structure of the graphs and the transformation rules are excluded, allowing us to improve the efficiency of the analysis of permutation equivalence.

Definition 8 (Dependency Net *DNet* of a derivation) Let $d = (d_1;\ldots;d_n)$ be a derivation respecting NACs of a GTS with NACs, let $STS_N(d) = (T,P,\hat{\pi})$ be the generated STS with NACs and let $s = seq(d) = \langle q_1,\ldots,q_n\rangle = \langle 1,\ldots,n\rangle$ denote the sequence of rule names in P according to the steps in d. The *dependency net* of d is given by the marked Petri net $DNet(d) = \langle N,M\rangle$, $N = \langle PL,TR,pre,post\rangle$, constructed by the steps in Fig. 5, where the steps are performed in the order they appear in the table.

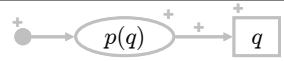
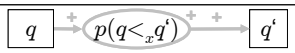
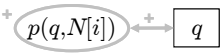
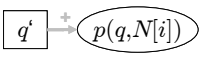
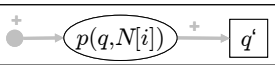
$\mathrm{STS}(d) = (T,P,\hat{\pi})$		$DNet(d) = ((PL,TR,pre,post),M)$
1. For each $q \in P$		
2. For all $q,q' \in P$, $q <_x q'$, $x \in \{rc,wc,d\}$		
3. For all $q \in P$ with $q \not<_{wdn[i]} q$, $i \in \mathbb{N}$		
	a) N[i] of q	
	b) For all $q' \in P$: $q' <_{wen[i]} q$	
	c) For all $q' \in P$: $q <_{wdn[i]} q'$	

Figure 5: Construction of the Dependency Net

Fig. 5 shows the steps of the construction of the dependency net. The steps are given as visualized rules, where gray line colour and plus-signs mark the inserted elements. In the first step they are created without context, but e.g. in step two the new place "$p(q <_x q')$" is inserted between the already existing transitions q and q'. The tokens of the marked Petri net are represented by bullets that are connected to their places by arcs. The first step creates a transition for each rule and the transition is connected to a marked place for ensuring that it cannot fire twice. In step 2, between each pair of transitions in each of the relations $<_{rc}$, $<_{wc}$ and $<_d$, a new place is created in order to enforce the corresponding dependency. The rest of the construction is concerned with places which correspond to NACs and can contain several tokens in general. Each token in such a place represents *the absence* of a piece of the NAC; therefore if the place is empty, the NAC is complete. In this case, by step (3a) the transition cannot fire. Consistently with this intuition, if $q' <_{wen[i]} q$, i.e. transition q' consumes part of the NAC $N[i]$ of q, then by step (3b) q' produces a token in the place corresponding to $N[i]$. Symmetrically, if $q <_{wdn[i]} q'$, i.e. q' produces part of NAC $N[i]$ of q, then by step (3c) q' consumes a token from the place corresponding to $N[i]$. Notice that each item of a NAC is either already in the start graph of the derivation or produced by a single rule. Furthermore, if a rule generates a part of one of its NACs, say $N[i]$ ($q <_{wdn[i]} q$), then by the acyclicity of $STS_N(d)$ the NAC $N[i]$ cannot be completed before the firing of q: therefore we ignore it in the third step of the construction of the dependency net.

A more formal definition of the construction, which explicitly defines the sets PL,TR, the pre and post mappings as well as the marking $M \in PL^{\oplus}$, is given by Def. 12 in [HCEK10]. Note

that the constructed net in general is not a safe one, because the places for the NACs can contain several tokens. Nevertheless it is a bounded P/T net, where the bound to the number of tokens is given by the maximum, taken over places representing NACs, of the number of rules that either weakly disable or weakly enable the specific NAC.

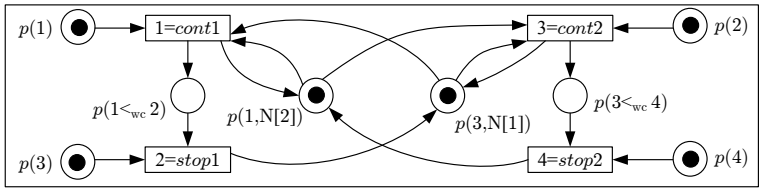


Figure 6: Dependency Net $DNet(d)$ as Petri Net

Example 4 (Dependency Net) *Consider the derivation d from Ex. 1 and its derived STS in Ex. 2. The marked Petri net shown in Fig. 6 is the dependency net $DNet(d)$ according to Def. 8. The places encoding the write causality relation are "$p(1 <_{wc} 2)$" and "$p(3 <_{wc} 4)$". For the NAC-dependencies we have the places "$p(1,N[2])$" for the second instantiated NAC in the first derivation step of d and "$p(3,N[1])$" for the third derivation step and its first instantiated NAC. The other two instantiated NACs are not considered, because the corresponding rules are weakly self-disabling ($q <_{wdn[i]} q$). At the beginning the transitions $cont1$ and $cont2$ are enabled. The firing sequences according to the derivations d and d' in Fig. 3 can be executed and they are the only firing sequences of this net. Thus, the net specifies exactly the derivations which are permutation-equivalent to d.*

We now show by Thm. 1 below that we can exploit the constructed Petri net $DNet(d)$ for a derivation d to characterise all derivations that are permutation-equivalent to d, by analysing the firing behaviour of $DNet(d)$. Note that according to Def. 8 each sequence s of rule names in the STS $STS_N(d)$ can be interpreted as a sequence of transitions in the derived marked Petri net $DNet(d)$, and vice versa. This correspondence allows us to transfer the results of the analysis back to the STS. More precisely, we can generate the set of all permutation-equivalent sequences by constructing the reachability graph of $DNet(d)$, which therefore can be considered as a compact representation of this equivalence class.

Recall that a *transition complete firing sequence* of a Petri net is a firing sequence where each transition of the net occurs at least once; notice also that in a dependency net according to Def. 8, each transition can fire at most once by construction. This means in our case each transition fires exactly once. The following Thm. 1 presents a sound and complete analysis of permutation equivalence by complete firing sequences in the corresponding dependency net.

Theorem 1 (Analysis of Permutation Equivalence of Derivations) *Let d be a derivation respecting NACs of a GTS with NACs, and let $DNet(d)$ be its dependency net. Then a derivation d' is permutation equivalent to d ($d' \overset{\pi}{\approx} d$) if and only if the sequence of names $s_{d'}$, which contains all the direct derivations of d in the order they are actually fired in d', is a transition complete firing sequence of the marked P/T Petri net $DNet(d)$.*

Proof (Sketch). Let d be a derivation with NACs, $STS_N(d)$ be its derived STS and $DNet(d)$ be the

constructed dependency net. We can interpret a transition complete firing sequence s of $DNet(d)$ within the STS $STS_N(d)$ and show that it corresponds to a valid derivation in $STS_N(d)$. This allows us to use Thm. 1 in [Her09] showing that the derivation derived from s is permutation-equivalent to d. Vice versa, given a derivation d', which is permutation-equivalent to d, we can show that the corresponding sequence $s_{d'}$ is a transition complete firing sequence in $DNet(d)$. For a complete proof see Cor. 1 in [HCEK10]. □

4 On the Cost of Analysis

Besides soundness and completeness of the analysis as presented in Thm. 1 we now focus on its efficiency. Therefore, we extend the previous example and compare the analysis efforts of the new technique with those of a direct analysis of the derivation. This comparison shows a significant advantage of the technique and the effect is not limited to specific examples. The benefit is high for transformation sequences, where many steps overlap on matches and include dependencies because of NACs. Clearly, if NACs are not involved permutation equivalence is equal to switch equivalence and in this case the reachability graph of the Petri net specifies all switch-equivalent derivations.

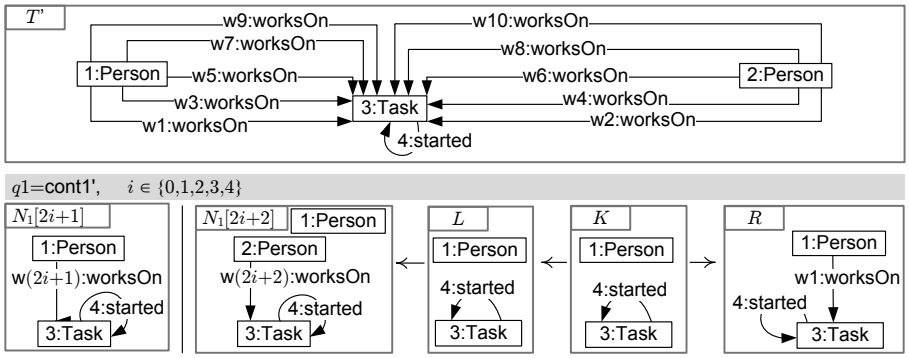


Figure 7: Part of the Derived STS $STS_N(\widetilde{d})$

Example 5 (Extended Derivation) *We extend the derivation d of Ex. 1 to a derivation $\widetilde{d}$, which specifies that the two persons are working on the same task, but they continue and stop their work five times, i.e. $\widetilde{d} = (d;d;d;d;d)$ is a derivation with 20 steps. The derived STS $STS(\widetilde{d})$ contains 20 rule occurrences and Fig. 7 shows its super object T' and the rule occurrence "cont1' " for the first step of $\widetilde{d}$. This rule occurrence has 10 NACs, one for each possible edge of type "worksOn" in T'. These NACs are visualised in the figure by two NACs with a parameter i ranging from zero to four. The derivation consists of 10 blocks of the form "contx; stopx". Each permutation-equivalent derivation of $\widetilde{d}$ has to preserve these blocks, otherwise a NAC would not be fulfilled or the causality relation would be violated. Thus there are $10! = 3.628.800$ permutation-equivalent derivations.*

Based on the dependency net $DNet(\widetilde{d})$ we can construct the reachability graph $RG(DNet(\widetilde{d}))$ for this marked Petri net with 20 transitions and 120 places. Each path in this graph specifies a permutation-equivalent derivation. An upper bound for the effort *eff* of constructing

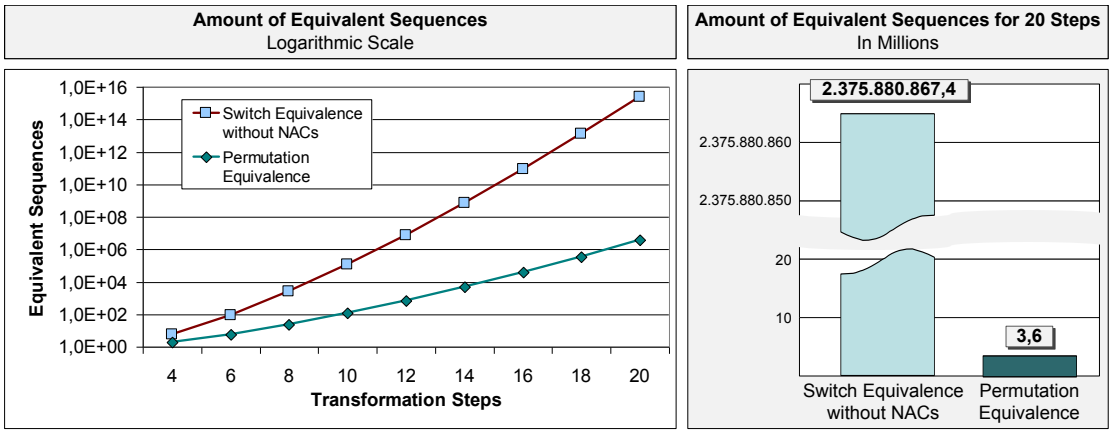


Figure 8: Comparison of the Amount of Equivalent Sequences

$RG(DNet(\widetilde{d}))$ is given by: $eff < 9 \cdot n$, where n is $n = 20 \cdot 10! = 72.576.000$, which is the number of derivation steps in the set of all permutation-equivalent derivations. The details of these and the following numbers are given in [HCEK10]. A direct generation of the permutation-equivalent derivations based on Def. 3 (brute force method) starts with a computation of the complete set of switch-equivalent derivations disregarding the NACs and thereafter the invalid ones are filtered out. This would lead to $F = 654.729.075$ times as many derivations as the number of permutation-equivalent derivations, because the blocks "$cont(x);stop(x)$" are split and many steps "$cont(x)$" can be shifted backwards. Thus, the lower bound for the brute force effort EFF is given by $F \cdot n \leq EFF$. In comparison we have for the effort eff of constructing the reachability graph of the dependency net:

$$\boxed{eff < 1{,}4 \times 10^{-8} EFF.}$$

Fig. 8 shows how the different amounts of equivalent sequences develop for 2 up to 10 blocks of "continue;stop" steps. Both analyses are fairly brute-force, since we did not integrate reductions, such as symmetry or partial-order reduction. However, the figures show that a definite gain in efficiency can be obtained, which we expect similarly also with additional reductions, which are mainly orthogonal to the reduction technique studied in this paper.

Of course, the effort for constructing the Petri net has also to be taken into account, but it does not significantly change the result. In general, the construction of the STS $STS(d)$ with its relations is shown to be of polynomial time complexity with respect to the length of the derivation d [Her09]. Furthermore, the construction of the dependency net is linear with respect to $STS_N(d)$ equipped with the derived relations and for this example contains only 120 places. Note that still all steps in $\widetilde{d}$ are sequentially dependent with NACs and therefore, no direct switching is possible.

5 Conclusion

In the framework of adhesive high-level replacement (HLR) systems there are many instantiations, such as graph transformation systems scaling up to typed attributed graph transformation systems with node type inheritance, and Petri net transformation systems - in particular for the

modelling of workflows of reconfigurable mobile adhoc networks [EHP+07, HEP07]. Each of them has its specific features, which support the modelling of systems in the concrete application domain. Negative Application Conditions (NACs) are an important control structure for these techniques and they are widely used for applications. However, the analysis of processes of such systems, i.e. the study of equivalence of derivations in the presence of NACs going beyond switch equivalence including NACs as studied in [HHT96, LEO06], was introduced only recently in [Her09]. This new notion of equivalence, called permutation equivalence, is studied in this paper. More precisely, we study the problem how to obtain, in an efficient way, all derivations d' which are permutation-equivalent to a given derivation d.

In order to provide a sound, complete and efficient analysis technique for permutation equivalence we have shown how the dependency net for the derivation can be constructed, which purely specifies the dependencies between the transformation steps including the inhibiting effects of the NACs. Based on the reachability graph of the dependency net we derive all valid permutations of the derivation steps of a given derivation d, i.e. the order of the applied rules together with the new matches. The derived derivations are exactly the permutation-equivalent derivations of d. While the example in this paper was kept compact, the overall approach can be applied to adhesive HLR systems in general, if suitable side conditions are fulfilled [HE08], which is the case for e.g. typed attributed graph transformation systems.

The efficiency of the Petri net approach is based on two advantages. First of all, the constructed Petri net only specifies the dependencies among the steps of the derivation, ignoring the concrete structure of the involved graphs: This advantage is independent of the presence of NACs. The second advantage is that NACs are respected during the generation of the permutation-equivalent sequences. Thus, the number of generated sequences during the analysis is reduced significantly if NACs are involved, as shown by the presented example.

Some of the problems addressed in this paper are similar to those considered in the process semantics [KK04] and unfolding [Bal00, BKS04] of Petri nets with inhibitor arcs, and actually we could have used some sort of inhibitor arcs to model the inhibiting effect of NACs in the dependency net of a derivation. However, we would have needed some kind of "generalised" inhibitor nets, where a transition is connected to several (inhibiting) places and can fire if at least one of them is unmarked. To avoid the burden of introducing yet another model of nets, we preferred to stick to a direct encoding of the process of a derivation into a standard marked P/T nets, leaving as a topic for future research the possible use of different models of nets for our dependency net.

Future work will encompass the extension of the presented technique to general application conditions in the form of nested application conditions [HP05, HP09], for which we already have concrete ideas. Further improvements of efficiency could be obtained by observing the occurring symmetries in the P/T Petri net, and applying symmetry reduction techniques on it. Additionally, the space complexity of the analysis could be reduced by unfolding the net and then representing all permutation-equivalent derivations in a more compact, partially ordered structure. We already implemented the construction of a dependency net from a given graph transformation derivation with NACs based on a recently developed graph transformation engine in Mathematica called AGT (Algebraic Graph Transformation) [BHE09].

Acknowledgements: The research was supported by the DFG project Behaviour-GT.

References

[Bal00] P. Baldan. *Modelling Concurrent Computations: from Contextual Petri Nets to Graph Grammars*. PhD thesis, Dipartimento di Informatica, Università di Pisa, 2000. Available as technical report n. TD-1/00.

[BCH+06] P. Baldan, A. Corradini, T. Heindel, B. König, P. Sobocinski. Processes for Adhesive Rewriting Systems. In *FoSSaCS'06*. LNCS 3921, pp. 202–216. Springer, 2006.

[BHE09] C. Brandt, F. Hermann, T. Engel. Security and Consistency of IT and Business Models at Credit Suisse realized by Graph Constraints, Transformation and Integration using Algebraic Graph Theory. In *Proc. Int. Conf. on Exploring Modeling Methods in Systems Analysis and Design 2009 (EMMSAD'09)*. LNBIP 29, pp. 339–352. Springer Verlag, Heidelberg, 2009.

[BKS04] P. Baldan, B. König, I. Stürmer. Generating Test Cases for Code Generators by Unfolding Graph Transformation Systems. In *Proc. of ICGT '04*. LNCS 3256, pp. 194–209. Springer, 2004.

[CHS08] A. Corradini, F. Hermann, P. Sobociński. Subobject Transformation Systems. *Applied Categorical Structures* 16(3):389–419, June 2008.

[EEPT06] H. Ehrig, K. Ehrig, U. Prange, G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. EATCS Monographs in Theoretical Computer Science. Springer, 2006.

[EHP+07] H. Ehrig, K. Hoffmann, J. Padberg, U. Prange, C. Ermel. Independence of Net Transformations and Token Firing in Reconfigurable Place/Transition Systems. In Kleijn and Yakovlev (eds.), *Proc. of ATPN'07*. LNCS 4546, pp. 104–123. Springer, 2007.

[HCEK10] F. Hermann, A. Corradini, H. Ehrig, B. König. Efficient Process Analysis of Transformation Systems Based on Petri nets (Long Version). Technical report 2010-3, Fak. IV, Technische Universität Berlin, 2010.
http://www.eecs.tu-berlin.de/menue/forschung/forschungsberichte/2010

[HE08] F. Hermann, H. Ehrig. Process Definition using Subobject Transformation Systems. *EATCS Bulletin* 95:153–163, 2008.

[HEP07] K. Hoffmann, H. Ehrig, J. Padberg. Flexible Modeling of Emergency Scenarios using Reconfigurable Systems. In *Proc. of IDPT'07*. Society for Design and Process Science, 2007.

[Her09] F. Hermann. Permutation Equivalence of DPO Derivations with Negative Application Conditions based on Subobject Transformation Systems. In Proc. of the Doctoral Symposium of ICGT'08. *Electronic Communications of the EASST* 16, 2009.

[HHT96] A. Habel, R. Heckel, G. Taentzer. Graph Grammars with Negative Application Conditions. *Fundamenta Informaticae* 26(3,4):287–313, 1996.

[HP05] A. Habel, K.-H. Pennemann. Nested constraints and application conditions for high-level structures. In *Proc. of Formal Methods in Software and System Modeling*. LNCS 3393, pp. 293–308. Springer, 2005.

[HP09] A. Habel, K.-H. Pennemann. Correctness of high-level transformation systems relative to nested conditions. *Mathematical Structures in Computer Science* 19(2):245–296, 2009.

[KK04] H. C. M. Kleijn, M. Koutny. Process semantics of general inhibitor nets. *Information and Computation* 190(1):18–69, 2004.

[Kre86] H.-J. Kreowski. Is parallelism already concurrency? Part 1: Derivations in graph grammars. In *Graph-Grammars and Their Application to Computer Science*. LNCS 291, pp. 343–360. Springer, 1986.

[LEO06] L. Lambers, H. Ehrig, F. Orejas. Conflict Detection for Graph Transformation with Negative Application Conditions. In *ICGT'06*. LNCS 4178, pp. 61–76. Springer, 2006.

[LS05] S. Lack, P. Sobociński. Adhesive and quasiadhesive categories. *Theoretical Informatics and Applications* 39(2):511–546, 2005.

Electronic Communications of the EASST
Volume of the Pre-proceedings of GT-VMT (2010)

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

Defining Models – Meta Models versus Graph Grammars

Berthold Hoffmann, Mark Minas

13 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

Defining Models – Meta Models versus Graph Grammars

Berthold Hoffmann[1], **Mark Minas**[2]

[1]hof@informatik.uni-bremen.de
Universität Bremen und DFKI Bremen, Germany

[2]Mark.Minas@unibw.de
Universität der Bundeswehr München, Germany

Abstract: The precise specification of software models is a major concern in model-driven design of object-oriented software. Metamodelling and graph grammars are apparent choices for such specifications. Metamodelling has several advantages: it is easy to use, and provides procedures that check automatically whether a model is valid or not. However, it is less suited for proving properties of models, or for generating large sets of example models. Graph grammars, in contrast, offer a natural procedure – the derivation process – for generating example models, and they support proofs because they define a graph language inductively. However, not all graph grammars that allow to specify practically relevant models are easily parseable. In this paper, we propose *contextual star grammars* as a graph grammar approach that allows for simple parsing and that is powerful enough for specifying non-trivial software models. This is demonstrated by defining program graphs, a language-independent model of object-oriented programs, with a focus on shape (static structure) rather than behavior.

Keywords: Graph grammar, Meta-model

1 Introduction

The precise specification of software models is a major concern in model-driven design of object-oriented software. Such specifications should support a checking procedure for distinguishing valid from invalid models, they should be well-suited for proofs in order to reason about the specified models, and they should allow for automatically generating model instances that may be used as test cases for computer programs being based on such models. The meta-modeling approach is an apparent choice for such specifications. It allows for precise model definitions and provides checking procedures. However, it is less suited for proofs and for instance generation.

Graph grammars are another natural candidate for specifying software models. They precisely define model languages, they are well-suited for proofs because of their inductive way of defining a graph language, and they offer a natural procedure for automatically generating model instances. Several kinds of graph grammars have been proposed in the literature. In order to allow for the specification of practically relevant models, we need a formalism that is *powerful* so that all properties of models can be captured, and *simple* in order to be practically useful, in particular for *parsing* models in order to determine their validity. However, easy to use graph grammar approaches often fail to completely specify models. As a case study, we consider *pro-*

gram graphs, a language-independent model of object-oriented programs that has been devised for specifying refactoring operations on programs [MEDJ05]. However, neither hyperedge replacement grammars [DHK97], nor the equivalent star grammars [DHJM10, Theorem 2.8], nor node replacement grammars [ER97] are powerful enough for completely specifying program graphs. Even the recently proposed adaptive star grammars [DHJ+06, DHJM10] fail for certain more delicate properties of program graphs. Their rules must be extended by application conditions in order to describe program graphs completely [Hof10].

In this paper, we propose the simpler graph grammar approach of *contextual star grammars*, an extension of star grammars that allows for easy parsing. Plain star rules are extended with positive and negative contexts, which must exist (or must not exist, respectively) in order to apply a star rule. Contexts may specify the existence of paths to certain nodes in the host graph, which may then be linked by the rule application. It turns out that program graphs can be defined by a contextual star grammar. Hence, this graph grammar approach allows for the precise specification of program graphs, i.e., non-trivial software models, supports a natural procedure for generating model instances, is well-suited for proofs, and allows for easy parsing. We contrast this grammar with the definition of program graphs using a conventional meta-model, which is specified by a UML class diagram and logical OCL constraints.

The paper is structured as follows. In Section 2, we recall how object-oriented programs can abstractly be represented as *program graphs*. We define the language of program graphs by a metamodel that consists of a class diagram with additional OCL constraints. Then we introduce star grammars in Section 3, show that they can define *program skeletons*, a sub-structure of program graphs, but fail to define program graphs themselves. We introduce contextual star grammars in Section 4, define program graphs with them, and outline an easy parsing procedure. We discuss these specifications—by metamodels and by contextual star grammars—in Section 5. We conclude with some remarks on related and future work in Section 6.

2 Graphs Representing Object-Oriented Programs

Program graphs have been devised as a language-independent representation of object-oriented code that can be used for studying refactoring operations [MEDJ05]. Therefore, they do not represent the abstract syntax of an object-oriented program, but rather its structural components and their dependencies. For instance, they capture single inheritance of classes and method overriding. Data flow between parameters, attributes, and method invocations represents the structure within method bodies.

Consider the object-oriented program shown in Figure 1 as an example. The program, written in object-oriented pseudo code, consists of class Cell and its subclass ReCell. The superclass has an attribute variable cts and two methods get and set. Subclass ReCell inherits these three features and additionally has an attribute variable backup and a method restore. Moreover, it overrides the method set of its superclass.

Figure 1 also shows the corresponding program graph. The graph is actually represented as an object diagram according to the program graphs' model whose class diagram is shown in Figure 2. Note that not all association roles of Figure 2 are shown Figure 1. Only one of the two roles of the associations is shown to avoid clutter. Note also the fat links in Figure 1; they

```
class Cell is
  var cts: Any;
  method get() Any is
    return cts;
  method set(var n: Any) is
    cts := n

subclass ReCell of Cell is
  var backup: Any;
  method restore() is
    cts := backup;
  override set(var n: Any) is
    backup := cts;
    super.set(n)
```

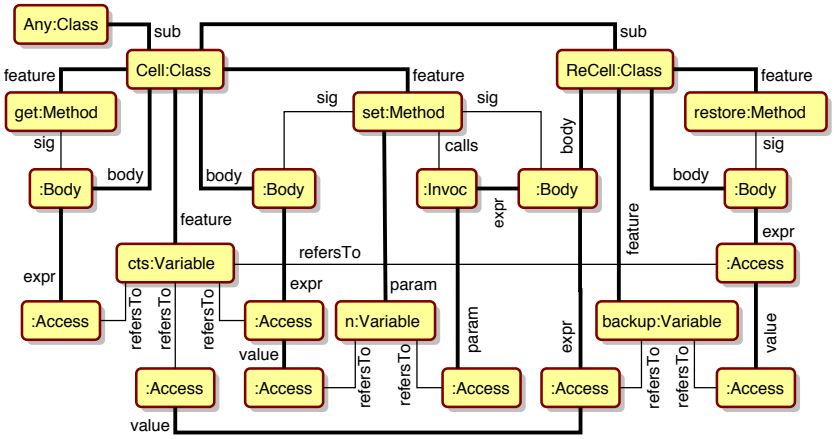


Figure 1: An object-oriented program and its program graph

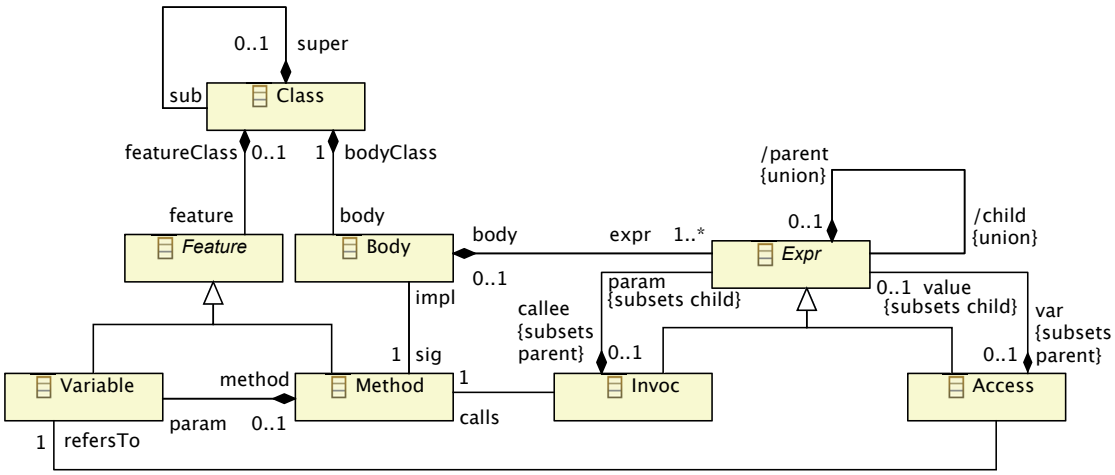


Figure 2: A model $\mathscr{M}$ for program graphs shown as a class diagram

correspond to the composition associations of Figure 2.

Each class is represented by a Class node. Note the universal superclass Any. Each class represents its (protected) attribute variables and (public) methods as features. Method nodes together with Variable nodes as their parameters represent method signatures; method bodies are represented separately by Body nodes. If a method is overridden, a new body refers to (we say: *implements*) the signature of the overridden method. Method set is an example: The signature node set:Method is implemented by two Body nodes, one being part of class Cell, the other being part of subclass ReCell. Data flow within method bodies is represented by (abstract) expressions that a body consists of (link expr). Expressions are represented by Access or Invoc nodes, both being subclasses of Expr. Access represents a reference to a variable either using its value, or assigning the value of an expression to it. Invoc nodes represent method invocations with their actual parameters being referred to by param links.

Figure 2 shows a UML class diagram for program graphs. The class diagram represents a model $\mathscr{M}$ of program graphs and also a meta-model because program graphs are models of object-oriented programs, i.e., $\mathscr{M}$ is a model of a modeling language. As usual, missing cardinalities mean 0..*. Also note the child-parent association at class Expr. It is subsetted by the corresponding associations (actually their association ends) for the subclasses Invoc and Access.

However, not all instances of the model represented by the class diagram are valid program graphs. Certain syntactic properties, usually called *static semantics* or *consistency conditions*, cannot be expressed by just a class diagram. The class $\mathscr{P}$ of all program graphs is rather defined by the class diagram and additional constraints:

Definition 1 (Program graphs) The class $\mathscr{P}$ of *program graphs* consists of all instances of the model $\mathscr{M}$ in Figure 2 that additionally satisfy the following constraints:

1. There is exactly one root class, i.e., class node without superclass.
2. A Variable node either belongs to a class (link feature) or to a method (link param).
3. An Expr node either belongs to a Body (link body) or to another expression (link parent).
4. A body b may implement a method contained in some class c if b is contained in c or in a subclass of c.
5. Every class may contain at most one body defining or overriding a particular method m.
6. An Access node e may refer to a Variable node representing an attribute contained in some class c if e is a sub-expression of a body that is contained in c or some subclass of c.
7. An Access node e may refer to a parameter of a Method node m if e is a sub-expression of a body implementing m.

```
context Class
def: visible : Set(Feature) =
if super→isEmpty() then
  feature
else
  feature→union(super.visible)
endif

context Expr
def: visible : Set(Feature) =
if body→isEmpty() then
  parent.visible
else
  body.bodyClass.visible
    →union(body.sig.param)
endif
```

```
1) inv uniqueRoot:
  Class.allInstances()
    →select(c | c.super→isEmpty())→size() = 1

2) context Variable inv validVariable:
  featureClass→isEmpty() <> method→isEmpty()

3) context Expr inv validExpr:
  body→isEmpty() <> parent→isEmpty()

4) context Body inv implementsVisibleMethod:
  bodyClass.visible→includes(sig)

5) context Body inv methodImplementedOnce:
  not bodyClass.body→exists(b | b <> self and b.sig = self.sig)

6,7) context Access inv accessesVisibleVariable:
    visible→includes(refersTo)
```

Figure 3: OCL constraints for the program graph model $\mathscr{M}$.

The *Object Constraint Language* OCL of the UML has been defined for formally defining such consistency conditions [Obj06]. Figure 3 shows the OCL constraints for program graphs based on the class diagram in Figure 2. The derived attributes visible of each Class instance contain all features directly defined in the own class together with all visible features of its superclass. These sets, together with all parameters of the implemented method, are propagated to sub-expressions of method bodies. Conditions 1–5 are formalized by constraints uniqueRoot, validVariable, validExpr, implementsVisibleMethod, and methodImplementedOnce, respectively. Constraint accessesVisibleVariable formalizes conditions 6 as well as 7. Numbers in Figure 3 correspond to the ones used above.

Note that conditions 1–3 require each node, except a unique Class node, to be a composite part of exactly one other node. The following observation follows from the fact that compositions cannot form cycles:

Fact 1 *The subgraph $\bar{P}$ of a program graph P induced by the composition edges is a spanning tree of P; the root of $\bar{P}$ is a Class node.*

3 Star Grammars

We first recall many-sorted graphs:

Definition 2 (Graph) Let $\Sigma = \langle \dot{\Sigma}, \bar{\Sigma} \rangle$ be a pair of disjoint finite sets of *sorts*.

A *many-sorted directed graph over* Σ (*graph*, for short) is a tuple $G = \langle \dot{G}, \bar{G}, s, t, \sigma \rangle$ where $\dot{G}$ is a finite set of *nodes*, $\bar{G}$ is a finite set of *edges*, the functions $s,t\colon \bar{G} \to \dot{G}$ define the *source* and *target* nodes of edges, and the pair $\sigma = \langle \dot{\sigma}, \bar{\sigma} \rangle$ of functions $\dot{\sigma}\colon \dot{G} \to \dot{\Sigma}$ and $\bar{\sigma}\colon \bar{G} \to \bar{\Sigma}$ associate nodes and edges with sorts.

Given graphs G and H, a pair $m = \langle \dot{m}, \bar{m} \rangle$ of functions $\dot{m}\colon \dot{G} \to \dot{H}$ and $\bar{m}\colon \bar{G} \to \bar{H}$ is a *morphism* if it preserves sources, targets and sorts.

Star grammars are a special case of double pushout (DPO) graph transformation [EEPT06]. By [DHJM10, Theorem 2.8], they are equivalent to hyperedge replacement grammars [DHK97] a well-understood context-free kind of graph grammars.

Definition 3 (Star) We assume that the node sorts contain *nonterminal sorts* $\dot{\Sigma}_{\mathrm{v}} \subseteq \dot{\Sigma}$ so that the *terminal node sorts* are $\dot{\Sigma}_{\mathrm{t}} = \dot{\Sigma} \setminus \dot{\Sigma}_{\mathrm{v}}$.

Consider a (star-like) graph X, with one center node c_X of nonterminal sort $x \in \dot{\Sigma}_{\mathrm{v}}$, and with some border nodes (of terminal sorts from $\dot{\Sigma}_{\mathrm{t}}$) so that the edges of X connect c_X to some of the border nodes. Then X is called an *incomplete star named* x. An incomplete star is called a *star* if each border node is incident with at least one edge. An (incomplete) star is *straight* if every border node is incident with at most one edge. Let $\mathscr{X}$ denote the class of *stars*, $\mathscr{G}(\mathscr{X})$ the graphs with stars, and $\mathscr{G}$ those without stars (where all nodes are labeled by $\dot{\Sigma}_{\mathrm{t}}$). We assume that nodes of nonterminal sort are not adjacent to each other in any graph.

Definition 4 (Star Replacement) An *incomplete star rule* is written $r = L ::= R$, where the *left-hand side* $L \in \mathscr{X}$ is a straight incomplete star and the *replacement* (right-hand side) is a graph

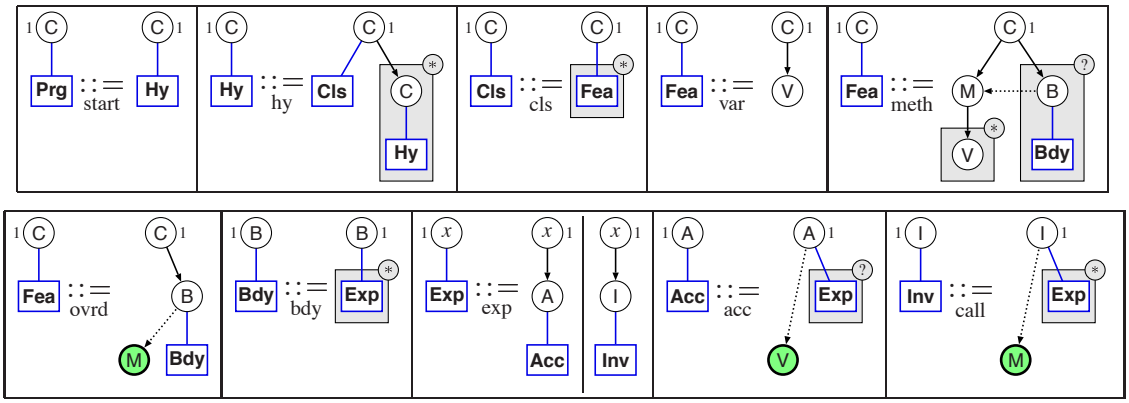


Figure 4: The rules of the star grammar PT

$R \in \mathscr{G}(\mathscr{X})$ that contains the border nodes of L. An incomplete star rule is called a *star rule* if L is a star.

The (incomplete) star rule r *applies* to some graph G if there is a morphism $m\colon L \to G$, yielding a graph H that is constructed by adding the nodes $\dot{R} \setminus \dot{L}$ and edges $\bar{R}$ disjointly to G, and by replacing, for every edge in $\bar{R}$, every source or target node $v \in \dot{L}$ by the node $\dot{m}(v)$, and by removing the edges $\bar{m}(\bar{L})$ and the node $\dot{m}(c_L)$.

We write $G \Rightarrow_r H$ if such a star replacement exists, $G \Rightarrow_{\mathscr{R}} H$ if $G \Rightarrow_r H$ for some r from a finite set $\mathscr{R}$ of (incomplete) star rules, and denote the reflexive-transitive closure of this relation by $\Rightarrow^*_{\mathscr{R}}$.

In the remainder of this section, we consider star rules only; incomplete star rules will only be used as a part of contextual star rules in Section 4.

Example 1 (Star Replacement) *Figure 4 shows a set of star rules. The center nodes of stars are depicted as boxes enclosing the star name. Nodes with terminal symbols are drawn as circles with their sort inscribed.*

The replacements of some rules show examples of abbreviating notation for repetitions in graphs and for optional subgraphs, which we will use frequently in star rules. Shaded boxes with a "∗" in the top-right corner, like those in rules hy, cls, meth, bdy, and call, designate multiple subgraphs *that may occur n times, $n \geqslant 0$, in the graph, with the same connections to the rest. If the shaded box has a "?" in its top-right corner (in rules meth and acc), its contained subgraph may occur 1 or 0 times in the graph. Note that this notation is similar to the EBNF notation of context-free string grammars. Likewise, it is just an abbreviating notation; one can always replace rules with this notation by several plain star rules using auxiliary stars and recursion, as long as shaded boxes do not include any of their rules' border nodes.*

Definition 5 (Star Grammar) $\Gamma = \langle \mathscr{G}(\mathscr{X}), \mathscr{X}, \mathscr{R}, Z \rangle$ is a *star grammar* with a *start star* $Z \in \mathscr{X}$. The *language* of Γ is obtained by exhaustive star replacement with its rules, starting from the start star: $\mathscr{L}(\Gamma) = \{G \in \mathscr{G} \mid Z \Rightarrow^*_{\mathscr{R}} G\}$.

Example 2 (Star Grammar for Program Skeletons) *Figure 4 shows the star rules generating the program skeletons that underlie program graphs. The rules define a star grammar* PT*, with the left-hand side of the first rule indicating the start star (named* **Prg***). Terminal node sorts are abbreviations of the class names in Figure 2, e.g.,* C *instead of* Class*. Terminal edge sorts are omitted completely. They can be easily inferred from the sorts of the incident nodes. Rule* exp*, which already is a shorthand for two rules, indicated by the two alternative replacements, uses x as border node sort where x stands for* B*,* A*, or* I*. Note that this is again just an abbreviating notation.*

Consider rules ovrd*,* acc*, and* call*, which generate method overriding, variable access, and method invocations. The overridden method, the accessed variable, and the invoked method, respectively, are distinguished by drawing them as filled circles with thick lines. In the skeleton rules, they are created anew. In a correct program graph according to Section 2, these distinguished nodes have to be identified ("merged") with the corresponding nodes that have been created elsewhere by rules* var *and* meth*, and represent their declarations. However, identification of nodes cannot be represented by star grammars, but requires contextual star grammars as defined in the next section.*

PT generates graphs that are closely related to program graphs. Given any graph $G \in \mathscr{L}(\mathsf{PT})$, let G^{T} denote the graph obtained from G by removing all filled nodes and dashed edges from G. Let $\mathscr{L}^{\mathrm{T}}(\mathsf{PT}) := \{G^{\mathrm{T}} \mid G \in \mathscr{L}(\mathsf{PT})\}$ denote the class of all such graphs. The following fact directly follows from the structure of rules in PT:

Fact 2 *$\mathscr{L}^{\mathrm{T}}(PT)$ is a language of trees.*

Obviously, grammar PT creates a single root class, and for each class an arbitrary number of subclasses. Each class gets arbitrarily many variables, method declarations, and bodies that consist of an arbitrary number of expressions that are either variable accesses or method invocations, consistent with the class diagram in Figure 2. Apparently, $\mathscr{L}^{\mathrm{T}}(\mathsf{PT})$ is the language of all trees that fit the class diagram when considering just its composition associations and additionally requiring the conditions 1–3 in Definition 1. Based on Fact 1 we can infer:

Fact 3 *The spanning tree $\bar{P}$ of each program graph $P \in \mathscr{P}$ (cf. Fact 1) is a member of $\mathscr{L}^{\mathrm{T}}(PT)$.*

Let $\mathscr{L}^{\mathrm{M}}(\mathsf{PT})$ denote the language of all graphs that can be obtained from a member of $\mathscr{L}(\mathsf{PT})$ by merging each filled node with a corresponding declaration node. Let $P \in \mathscr{P}$ be an arbitrary program graph. Its spanning tree $\bar{P}$ is equal to graph G^{T} of some graph $G \in \mathscr{L}(\mathsf{PT})$ because of Fact 3. P can be obtained by identifying the filled nodes of G with appropriate declaration nodes, i.e., $P \in \mathscr{L}^{\mathrm{M}}(\mathsf{PT})$. On the other hand, it is obvious that each identification of filled nodes with declaration nodes fits the class diagram and conditions 1–3 in Definition 1. Therefore, each graph $G \in \mathscr{L}^{\mathrm{M}}(\mathsf{PT})$ satisfying conditions 4–7, too, is a program graph. We can summarize:

Fact 4 *The set of all graphs $G \in \mathscr{L}^{\mathrm{M}}(PT)$ that satisfy conditions 4–7 in Definition 1 is equal to $\mathscr{P}$.*

Star grammars are context-free in the sense defined by Courcelle [Cou87]. This suggests that

their generative power is limited. Indeed, we have the following

Fact 5 *There is no star grammar* Γ *with* $\mathscr{L}(\Gamma) = \mathscr{P}$.

Proof Sketch *Consider the rule* call *in Figure 4. (The situation is similar for rules* ovrd *and* acc.*) This rule generates a new node for a method (drawn filled, and with thick lines). However, for generating a program graph, the rule should insert a call to a method that already exists in the host graph, and may be called in the expression. Due to the restricted form of star rules, the method had to be on the border of the rule. Since expressions may call every method in the graph, the star rule must have all these methods as its border nodes so that one of them can be selected. However, the number of callable methods depends on the size of the program, and is unbounded. Thus a finite set of star rules does not suffice to define all legal method calls.* □

4 Contextual Star Grammars

The adaptive star grammars defined in [DHJ+06] overcome the deficiencies illustrated in the proof sketch for Fact 5 by allowing the left-hand sides of star rules to adapt to as many border nodes as needed. A further extension, by positive and negative application conditions, extended their power, however with rather complicated rules [Hof10]. In this paper, we therefore consider a different extension of star rules with which program graphs can be completely defined in a simple way. We allow that the application of a star rule depends on its *context* in the host graph. Some *positive* context may be required whereas other, *negative*, context is forbidden if the rule shall apply. Nodes of the positive context may be used by the replacement graph of the rule.

Definition 6 (Contextual Star Rule) A *contextual star rule* r has the form $r = P/N \wr L ::= R$, where $L ::= R$ is an incomplete star rule, and the *positive contexts* P as well as the *negative contexts* N are (decidable) sets of graphs that contain the border nodes of L as subgraph. (Note that r is a star rule if L is a star and the sets P and N are empty.)

The contextual star rule r *applies* to some graph G if there is a morphism $m\colon L \to G$ that can be extended to a morphism $C \to G$ for at least one positive context $C \in P$ (if $P \neq \emptyset$) and that cannot be extended to a morphism $C \to G$ for any negative context $C \in N$, yielding a graph H by applying the incomplete star rule $L ::= R$ to G with morphism m.

Then we write $G \overset{c}{\Longrightarrow}_r H$, $G \overset{c}{\Longrightarrow}_{\mathscr{R}} H$ if $G \overset{c}{\Longrightarrow}_r H$ for some r from a finite set $\mathscr{R}$ of contextual star rules, and denote the reflexive-transitive closure of this relation by $\overset{c}{\Longrightarrow}^*_{\mathscr{R}}$.

Definition 7 (Contextual Star Grammar) $\Gamma = \langle \mathscr{G}(\mathscr{X}), \mathscr{X}, \mathscr{R}, Z \rangle$ is a *contextual star grammar* (CSG) with a *start star* $Z \in \mathscr{X}$ and a finite set $\mathscr{R}$ of contextual star rules. The *language* of Γ is obtained by exhaustive application of its rules, starting from the start star: $\mathscr{L}(\Gamma) = \{G \in \mathscr{G} \mid Z \overset{c}{\Longrightarrow}^*_{\mathscr{R}} G\}$.

In the following, we will either enumerate context graphs of the sets P and N of positive and negative contexts, respectively, or we will specify them by path expressions similar to the PROGRES language [Sch97]. Even more powerful specifications of context graphs are conceivable, e.g., by hyperedge replacement systems, as proposed in [HR10], or by star grammars. But this

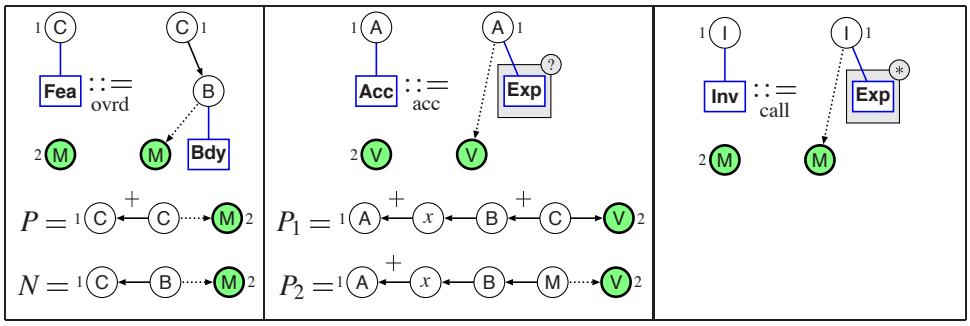


Figure 5: The contextual star rules of the grammar PG

is not necessary for specifying program graphs.

Example 3 (Contextual Star Grammar for Program Graphs) *The CSG for program graphs contains the star rules start, hy, cls, var, meth, bdy, and exp of PT in Figure 4, and the contextual rules in Figure 5. In these rules, specifications of positive and negative contexts are drawn below their incomplete star rules. Small numbers indicate the correspondence between context nodes and border nodes.*

So filled nodes in rules ovrd, acc, and call of PT are turned into context nodes in PG. Path expressions encode conditions 4–7 in Definition 1. A small "+" above an edge represents any path of length $\geqslant 1$. In rule ovrd, the path expression of the positive context P encodes condition 4, i.e., the method declaration must be inherited by the current class, while the negative context encodes condition 5, i.e., the current class must not have a second body for the same method declaration. Rule acc does not have a negative context, but two positive contexts. Rule acc may be applied either with context P_1 or with context P_2. P_1 and P_2 encode conditions 6 and 7, i.e., access to a visible attribute variable and to a visible parameter, respectively. The node labeled x represents a node of any sort. Rule call has empty positive and negative contexts and is applicable if its incomplete star rule applies. However, it still is a contextual star rule because its left-hand side requires the existence of an M-node in the context.

In Figure 6, some steps in the derivation of the program graph in Figure 1 are shown. The first graph represents the class hierarchy of the program graph. The grey bubbles in these graphs abstract from parts of the derived graph that are nor relevant for the derivation steps shown. The rule ovrd can be applied to this graph, where we draw the context node filled, with thick lines, and underlay the path leading to it in grey. We use the same drawing convention for the remaining derivation steps using bdy, call, acc, and again acc.

The following fact is a direct implication of Fact 4:

Fact 6 $\mathscr{L}(PG) = \mathscr{P}$.

Star grammars can be easily transformed into equivalent hyperedge replacement grammars [DHJM10, Theorem 2.8] and vice versa by interpreting stars as hyperedges with nonterminal labels. Hence, parsers for hyperedge replacement grammars like the DIAGEN parser [Min02]

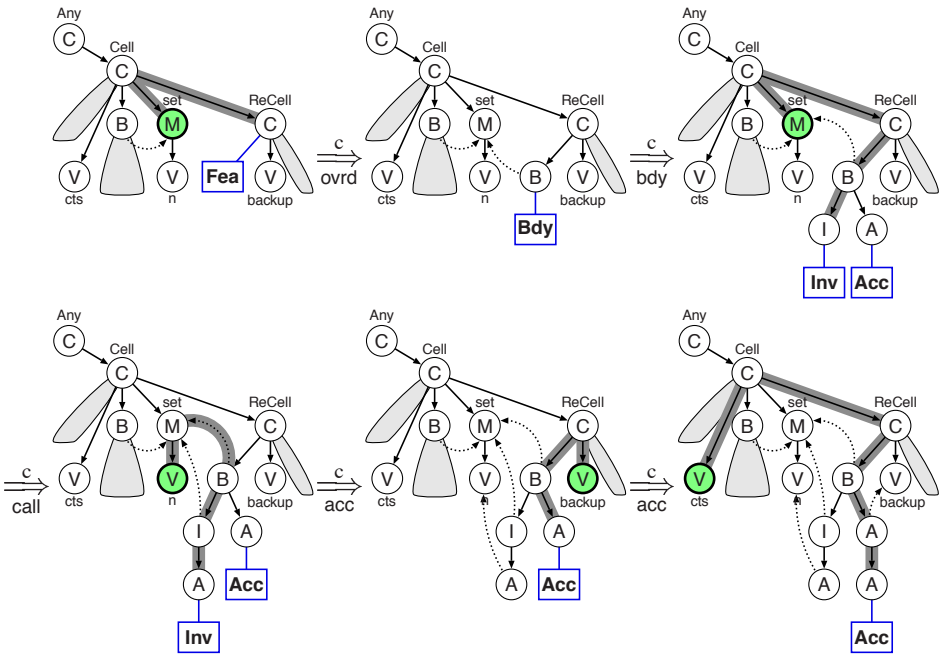


Figure 6: Deriving the program graph of Figure 1 with PG

can be used to parse star grammars. The same parser with only slight extensions can also be used for parsing CSGs. This parser is outlined below.

The parser works on CSGs like a *Cocke-Younger-Kasami* (CYK) parser for string grammars. The CSG has to be in *Chomsky normalform* (CNF) so that each production rule is either terminal or nonterminal. The right-hand side of a terminal rule is a terminal graph with only one edge, the right-hand side of a non-terminal rule is the union of two stars. Similar to string grammars, each CSG that does not produce the empty graph can be transformed into CNF. Given a terminal graph G, the parser creates a derivation for G, if it exists, in two phases. The first phase completely ignores contexts of contextual star rules and creates candidates for derivations. The second phase searches for a derivation by checking these candidates, this time considering contexts.

In the first phase, the parser builds up n sets $L_1, L_2, \ldots, L_n$ where n is the number of edges in G. Each set L_i eventually contains all stars that can be derived to any subgraph of G that contains exactly i edges. Set L_1 is built by first finding each embedding of the right-hand side of each terminal rule and adding the star of the corresponding left-hand side to L_1. If the corresponding rule is a contextual star rule with an incomplete star as left-hand side, only the (complete) star within the incomplete star is added to L_1. The remaining sets are then constructed using nonterminal rules. A nonterminal rule is reversely applied by searching for appropriate stars s and s' in sets S_i and S_j, respectively. If a nonterminal rule (without considering contexts) is applicable, i.e., if the rule's right-hand side is isomorphic to the union of s and s', a new star corresponding to the rule's left-hand side is added to a set S_k. Note that $k = i + j$ since each star in a set S_i can be derived to a subgraph of G with exactly i edges. Each instance of the start star Z in S_n represents a derivation candidate for G.

The second parser phase checks these derivation candidates by testing for each application of a contextual star rule whether required context has already been created and forbidden context has not (yet) been created earlier in the derivation. The parser stops when it finds the first valid derivation, or when it has checked the last derivation candidate without success.

5 Discussion

In the previous sections we have used two different techniques to describe program graphs. We have shown that the specification of program graphs by CSGs is indeed equivalent to their definition using a model together with OCL constraints. Moreover, both approaches allow for automatic checking whether a given graph is a valid program graph. Both specifications are actually even more closely related as the following discussion shows.

The CSG for program graphs consists of (plain) star rules and contextual star rules. As described by Fact 2, the tree structure (made from composition links) of program graphs can be constructed by (plain) star rules. Plain star grammars, however, fail for links refersTo and call, i.e., those program graph edges that represent references to objects that are located "far away" in the program structure. Contextual star rules are needed to add those edges; the far away objects are represented by the rules' context nodes (Figure 5). The positive and negative contexts of those rules play the same role like the OCL constraints for conditions 4–7 in Figure 3. These constraints actually can be considered as an OCL implementation of the rules' context conditions. E.g., constraint methodImplementedOnce for condition 5 in Definition 1 prohibits a second body of the same method within the same class; this exactly corresponds to the negative context N of rule ovrd (Figure 5). The path expressions P, P_1, and P_2 of rules ovrd and acc, respectively, realize conditions 4, 6, and 7. Their OCL realizations make use of the derived attributes visible whose recursive definition exactly reflects the iteration operator "+" in the path expressions.

The contextual star grammar PG (Figure 4 and 5) for program graphs has been created by hand. The discussion, however, has revealed a close relation between OCL constraints and positive as well as negative contexts on the one hand, and between class diagram and contextual star rules without those contexts on the other hand. This close relation may lead to a procedure for translating CSGs and meta models with OCL constraints into each other at least in a semi-automatic way. This line of research is also motivated by work of Ehrig et al. [EKT09]. They create graph transformation rules with graph constraints from meta models with OCL constraints. The graph transformation system is then used to automatically generate model instances of the meta model. However, they only consider very restricted OCL constraints that are not sufficient for the specification of program graphs, and their generated layered graph transformation rules are rather complicated, and are less suited for parsing.

6 Conclusions

We have extended star rules by positive and negative context. These contextual star grammars allow to define program graphs precisely, without sacrificing parseability. Program graphs, which represent certain aspects of object-oriented programs, can also be defined by a class diagram together with OCL constraints in a model-based style. However, this approach is less suited

for inductive proofs or for automatic instance generation than the proposed grammar-based approach. A comparison of both specification approaches, however, has shown close relations between both specification approaches which may allow for a semi-automatic translation process between both specification approaches. This will be subject of future work.

Too many kinds of graph grammars are related to contextual star grammars to mention all of them. So we restrict our discussion to those that aim at similar applications. Contextual star rules re-use *path expressions* first devised by M. Nagl [Nag79], which have later been implemented in the PROGRES graph transformation language [Sch97]. Using context conditions and examining their relation to logical graph properties and constraints is not new either. A. Habel and K.-H. Pennemann have shown that nested graph conditions are equivalent to first-order graph formulas [HP09]. These conditions are still too weak to specify program graphs, as they only allow to require or forbid the existence of subgraphs of bounded size. Only in [HR10], A. Habel and H. Radke devised nested graph conditions with variables that allow to express the conditions of CSGs (and more). However, rules and grammars using such conditions are not yet considered in that paper. Context-embedding rules [Min02] extend hyperedge replacement grammars by rules that add a single edge to an arbitrary graph pattern. They are used to define and parse diagram languages, but are not powerful enough to define models like program graphs. Graph reduction grammars [BPR10] have been proposed to define and check the shape of data structures with pointers. While the form of their rules is not restricted, reduction with them is required to be terminating and confluent, so that random application of rules provides a backtracking-free parsing algorithm. It is still open whether graph reduction grammars suffice to define program graphs.

The nested patterns [HJG08] recently introduced to GRGEN [GBG+06], an efficient graph rewrite tool, allow to express nested graph conditions with variables that are defined by hyperedge replacement systems, and can thus be used to implement contextual star grammars (like that for program graphs) in the future.

Acknowledgements: We thank the reviewers for their constructive criticism that helped to improve our paper.

Bibliography

[BPR10] A. Bakewell, D. Plump, C. Runciman. Specifying Pointer Structures by Graph Reduction. *Mathematical Structures in Computer Science*, 2010. To appear.

[CEM+06] A. Corradini, H. Ehrig, U. Montanari, L. Ribeiro, G. Rozenberg (eds.). *3rd Int'l Conference on Graph Transformation (ICGT'06)*. Lecture Notes in Computer Science 4178. Springer, 2006.

[Cou87] B. Courcelle. An Axiomatic Definition of Context-free Rewriting and its Application to NLC rewriting. *Theoretical Computer Science* 55:141–181, 1987.

[DHJ+06] F. Drewes, B. Hoffmann, D. Janssens, M. Minas, N. V. Eetvelde. Adaptive Star Grammars. Pp. 77–91 in [CEM+06].

[DHJM10] F. Drewes, B. Hoffmann, D. Janssens, M. Minas. Adaptive Star Grammars and Their Languages. *Theoretical Computer Science*, 2010. Accepted for publication.

[DHK97] F. Drewes, A. Habel, H.-J. Kreowski. Hyperedge Replacement Graph Grammars. Chapter 2 in [Roz97].

[EEPT06] H. Ehrig, K. Ehrig, U. Prange, G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. EATCS Monographs on Theoretical Computer Science. Springer, 2006.

[EKT09] K. Ehrig, J. M. Küster, G. Taentzer. Generating instance models from meta models. *Software and System Modeling* 8(4):479–500, 2009.

[ER97] J. Engelfriet, G. Rozenberg. Node Replacement Graph Grammars. Chapter 1 in [Roz97].

[GBG+06] R. Geiß, G. V. Batz, D. Grund, S. Hack, A. Szalkowski. GrGen: A Fast SPO-Based Graph Rewriting Tool. Pp. 383–397 in [CEM+06]. URL: http://www.grgen.net.

[HJG08] B. Hoffmann, E. Jakumeit, R. Geiß. Graph Rewrite Rules with Structural Recursion. In Mosbah and Habel (eds.), *2nd Intl. Workshop on Graph Computational Models (GCM 2008)*. Pp. 5–16. 2008.

[Hof10] B. Hoffmann. Conditional Adaptive Star Grammars. *Electr. Comm. of the EASST*, 2010. To appear.

[HP09] A. Habel, K.-H. Pennemann. Correctness of high-level transformation systems relative to nested conditions. *Mathematical Structures in Computer Science* 19(2):245–296, 2009.

[HR10] A. Habel, H. Radke. Expressiveness of graph conditions with variables. In Ehrig and Ermel (eds.), *International Colloquium on Graph and Model Transformation (GraMoT'10)*. 2010. To appear in Electr. Comm. of the EASST.

[MEDJ05] T. Mens, N. V. Eetvelde, S. Demeyer, D. Janssens. Formalizing refactorings with graph transformations. *Journal on Software Maintenance and Evolution: Research and Practice* 17(4):247–276, 2005.

[Min02] M. Minas. Concepts and Realization of a Diagram Editor Generator Based on Hypergraph Transformation. *Science of Computer Programming* 44(2):157–180, 2002.

[Nag79] M. Nagl. *Graph-Grammatiken: Theorie, Anwendungen, Implementierungen*. Vieweg-Verlag, Braunschweig, 1979. In German.

[Obj06] Object Management Group. Specification of the Object Constraint Language. http://www.omg.org/spec/OCL/2.0/, 2006.

[Roz97] G. Rozenberg (ed.). *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. I: Foundations*. World Scientific, Singapore, 1997.

[Sch97] A. Schürr. Programmed Graph Replacement Systems. Chapter 7 in [Roz97].

Electronic Communications of the EASST
Volume of the Pre-proceedings of GT-VMT (2010)

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

Specifying and Generating Editing Environments for Interactive Animated Visual Models

Torsten Strobl and Mark Minas

13 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

Specifying and Generating Editing Environments for Interactive Animated Visual Models

Torsten Strobl[1] **and Mark Minas** [2]

[1] Torsten.Strobl@unibw.de
[2] Mark.Minas@unibw.de
Computer Science Department
Universität der Bundeswehr München
85577 Neubiberg, Germany

Abstract: The behavior of a dynamic system is most easily understood if it is illustrated by a visual model that is animated over time. Graphs are a widely accepted approach for representing such dynamic models in an abstract way. System behavior and, therefore, model behavior corresponds to modifications of its representing graph over time. Graph transformations are an obvious choice for specifying these graph modifications and, hence, model behavior. Existing approaches use a graph to represent the static state of a model whereas modifications of this graph are described by graph transformations that happen instantaneously, but whose durations are stretched over time in order to allow for smooth animations. However, long-running and simultaneous animations of different parts of a model as well as interactions during animations are difficult to specify and realize that way. This paper describes a different approach. A graph does not necessarily represent the static aspect of a model, but rather represents the currently changing model. Graph transformations, when triggered at specific points of time, modify such graphs and thus start, change, or stop animations. Several concurrent animations may simultaneously take place in a model. Graph transformations can easily describe interactions within the model or between user and model, too. This approach has been integrated into the DIAMETA framework that now allows for specifying and generating editing environments for interactive animated visual models. The approach is demonstrated using the game *Avalanche* where many parallel and interacting movements take place.

Keywords: animated visual language

1 Introduction

Visual modeling is already one of the most useful techniques for describing complex systems. There is the need for many different, also domain-specific visual languages, each appropriate for dedicated purposes. Meta-tools like DIAGEN/DIAMETA [Min06], GenGED [Erm06] or AToM^3 [LV02] help specifying such languages and generating corresponding editors.

However, visual languages are not limited to static models. When modeling dynamic systems, dynamic models can be used to simulate the system and help understanding it. Smooth animations can even improve this visual comprehension. Suitable visual languages for animations span

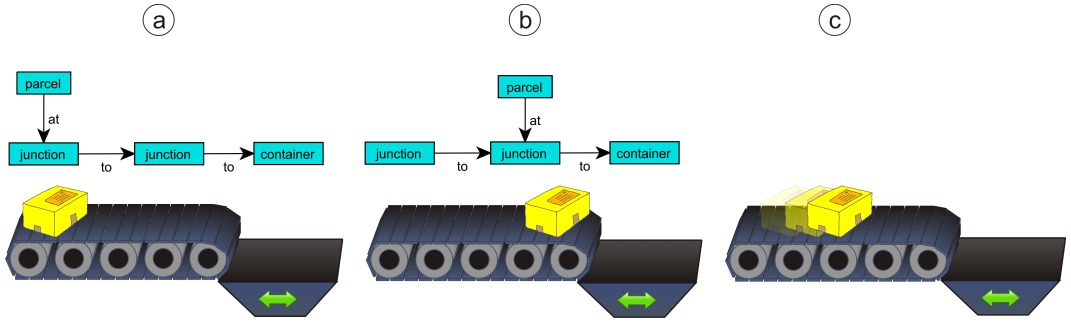


Figure 1: Existing approach: before GT (a), after GT (b) and animated scene (c)

formal/mathematical models like petri nets, educational languages like Alligator Eggs [SM09] and even highly interactive programming languages like the Alternate Reality Kit [Smi86].

It is a common approach to use graphs for representing such models in an abstract way. The model is changed by transforming the underlying graph via graph transformations (GT). Existing techniques associate each graph (the graph before and after the GT) with a static model and therefore static visualization. An example in the domain of conveyor systems is shown in Figure 1. While (a) shows the graph and its visual representation before the GT, (b) shows them afterwards. In order to avoid a jumping parcel in the visualization, the instantaneous GT can be stretched over time and a smooth animation is applied to the state transition (c).

This approach comes along with some obvious problems. If the system behavior includes multiple, independent animations in different parts of the model at the same time, specification becomes difficult. The problem becomes even more complicated if multiple animations overlap in time or if interactive environments shall be realized that way. As an instance consider the conveyor system where the container may move, too, and those movements might be triggered by user interactions. If such a user interaction takes place while a parcel is on its way as shown in Figure 1c, the system should immediately stop the conveyor with the parcel at the current position. However, this situation cannot be represented by the chosen graph.

This paper describes another approach: graphs represent the currently changing model and GTs are used to start, stop and modify animations. The rest of the paper is structured as follows: Section 2 describes *Avalanche* as a motivating example. Section 3 introduces the abstract animation system, which serves as abstract formal system of the described approach in Section 4 using GTs. As an example, Section 5 shows an application of this approach in order to specify *Avalanche*. Section 6 outlines related work, and Section 7 concludes the paper.

2 Avalanche

Originally, *Avalanche* is a board game for two and more players. It was republished by different publishers and is available under different names and variants. For this paper, we concentrate on the main game mechanics and ignore objectives and other gaming aspects. The following paragraphs describe an *Avalanche* variant that is suitable as an exemplary dynamic system.

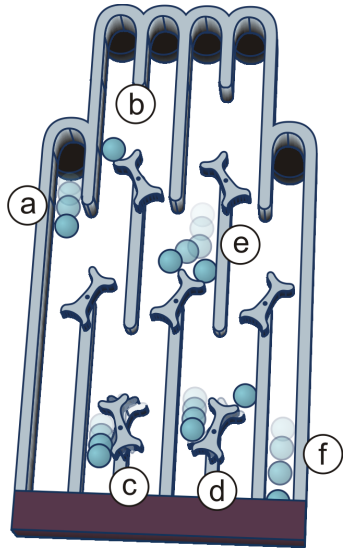


Figure 2: *Avalanche* board

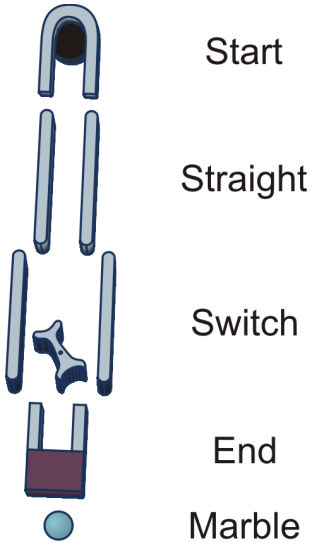


Figure 3: *Avalanche* pieces

The board of the game is an inclined plane. On this plane, there are multiple lanes, which are directed top-down. However, lanes can be interrupted by switches. Switches are placed in between two adjacent lanes. In this area, there is no border between left and right lane, and the switch can be tilted freely to the left or right. Shifted to one side, the switch can block the direct top-down way of the corresponding lane. Figure 2 shows an exemplary *Avalanche* board.

The game is played by putting marbles on the start (top) of a lane. After putting a marble there, it rolls down along the lane. Figure 2 (a) shows a position where a marble can be brought into play, and the marble starts rolling. While rolling down, a marble can be stopped by a switch that is facing the marble's lane (b). Rolling along the opposite lane, the marble hits the bottom of the switch, and therefore tilts the switch to the other side like in (c). This feature can be used for releasing a marble that has been stopped by the switch. After this marble has been released, it continues to roll along its lane, as shown in (d). In this concrete situation, the switch is lying on the left side afterwards[1]. If a marble hits another marble that has been stopped by a switch like in (e), the rolling marble changes the lane, continues rolling, and releases the other, previously blocked marble. Finally, when a marble reaches the end (bottom) of its lane as shown in (f), it is removed from the board.

Each *Avalanche* board consists of four types of building blocks (see Figure 3): *Start* (starts each lane; marbles can be placed there), *Straight* (straight lane), *Switch* (can block rolling marbles and can be switched by rolling marbles; each switch can be in the left or right position), and *End* (end of each lane; marbles are taken out of the game there).

[1] Depending on the *Avalanche* variant, it is also possible that the released marble immediately triggers the switch again. If this is the case, the switch would be lying on the right side afterwards.

3 Abstract Animation Systems

The *Avalanche* game is a continuous dynamic system. In order to describe its behavior, it is a common approach to look on it as a discrete event system with specified events for the collision of marbles with switches, putting new marbles on the field, etc. In the time between these events, the marbles are moving over the board, which can be illustrated by an animated visual model. This section formalizes *abstract animation systems* (ASSs), a particular kind of event-oriented systems, which is especially suited for interactive and animated visual languages. It is an abstraction of the animation approach using graphs and GTs described in the following sections.

The idea of defining a visual animated model by an ASS is to specify a state-transition-system that performs state transitions triggered by events at certain points in time. The visual representation of a model is determined by the current state of the state-transition-system and the current time. That means, the visual representation just depends on the (continuously) proceeding time between two consecutive state transitions. State transitions are triggered by events. Events may have an external source, e.g., the user placing a marble at a *Start* piece. These *external events* may happen at any time. Other events, called *internal events*, happen because of the current state after a certain span of time. For instance, if a switch starts switching from right to left, the switch will hit its left border after a fixed span of time. For this purpose, an event *Switching-CompleteLeft* is triggered, which stops the switching (cf. Section 5). Models usually consist of different parts, each of them with more or less independent behavior (e.g., an *Avalanche* board with several switches and marbles). States and events must appropriately reflect this composite structure.

In the following, ASSs are introduced more formally. Let T represent the absolute time. For each point in time $t \in T$, let $\omega > t$ and let $T^\omega = T \cup \{\omega\}$. For any set X, the power set of X is denoted by $\mathbf{P}(X)$.

Definition 1 An *abstract animation system* is a tuple $A = (S, E, \tilde{E}, s_0, \delta, \tau, \varepsilon)$. S is the *set of states* and s_0 is the *initial state*, $s_0 \in S$. E is the *set of all events*, whereas $\tilde{E} \subseteq E$ is the *set of internal events*. $E \setminus \tilde{E}$ denotes the set of *external events*. δ is the *state transition function*, $\delta : S \times E \times T \to \mathbf{P}(S)$. τ and ε describe the *occurrence of internal events*, $\tau : S \to T^\omega$ and $\varepsilon : S \to \mathbf{P}(\tilde{E})$. Each of these sets may be uncountably infinite.

The abstract animation system A is started in state s_0 at some point in time t_0. The execution of A is expressed by the chronology of occurred states; state changes are triggered by occurring events (internal and also external) at certain points in time. Each execution can be described by a *trace*

$$s_0 \xrightarrow[t_1]{e_1} s_1 \xrightarrow[t_2]{e_2} s_2 \xrightarrow[t_3]{e_3} \cdots \xrightarrow[t_i]{e_i} s_i \xrightarrow[t_{i+1}]{e_{i+1}} \cdots$$

of assumed states $s_0, s_1, s_2, \ldots \in S$. In each case, at the point in time t_i ($i > 0$), event e_i occurs and triggers the state transition from s_{i-1} to $s_i \in \delta(s_{i-1}, e_i, t_i)$. Either $e_i \in \varepsilon(s_{i-1})$, i.e., e_i is an internal event, then $t_i = \tau(s_{i-1})$, or $e_i \in E \setminus \tilde{E}$, i.e., e_i is an external event, then $t_i \in [t_{i-1}, \tau(s_{i-1})]$.

This definition of state transitions reflects the motivation of ASSs at the beginning of this section: If the state-transition-system is in a certain state, an internal event will happen after a certain

span of time. This is reflected by function ε and τ. However, an external event may happen at any time, i.e., possibly before the scheduled internal event. The external event triggers a state transition, and the previously scheduled internal event may be re-scheduled again, specified by functions δ, ε, and τ.

Following the motivation at the beginning of this section, the visual representation of an animated visual model with an ASS A is a *visualization function* $I : S \times T \to R$ where R represents all possible graphical illustrations. Given a trace $s_0 \xrightarrow[t_1]{e_1} s_1 \xrightarrow[t_2]{e_2} s_2 \cdots$ of A, the visual representation of the model at any point in time t is $I(s,t)$ where s is the current state at t, i.e., $s = s_i$ for an appropriate i such that $t \in [t_i, t_{i+1})$.

4 Animations using Graphs and Graph Transformations

The ASS formalism has been introduced to clarify the concepts determining the behavior of visual animated models. It can be used for describing an animated system like *Avalanche*, but it is less suited for actually specifying concrete animated languages. Instead, we use the widely spread approach of typed graphs for internally representing visual models. Existing meta-tools like DIAMETA [Min06], which are based on this approach (we do not distinguish plain graphs from hypergraphs here), then allow for generating editing environments from the visual language specifications. These tools already represent the static structure of a visual model by typed attributed graphs; the visual representation of a model is just a view of this graph. We now augment these graphs such that they also represent the current model state according to the notion of ASSs. The visualization function, as described in the previous section, again provides a view of the graph. However, it must take the continuously proceeding time into account when determining the visual representation of the animated model.

Because graphs are used for representing the ASS state, state transitions correspond to graph modifications. GTs are an obvious choice to perform and specify these modifications. In order to realize the state transition function δ of the ASS using GTs, each event (type) is associated with a particular set of graph transformation rules. Whenever an event occurs, especially if this is an external event sent to the system, the associated rules are selected for application in order to change the graph and state of the system, resp. If an event carries additional information, e.g., the *Start* piece where the user has placed a new marble, this information must be passed as a partial embedding morphism to the selected GT rules.

While external events are generated outside (e.g., by user interaction) and sent to the system by selecting appropriate GTs, scheduling internal events must be calculated by the animated system based on its state, i.e., functions ε and τ must be specified. This is done in the following way: Instead of directly calculating the next internal event and the point in time when it has to be triggered, this is done for each part of the whole system. The next scheduled internal event then is just the one of all calculated events that happens first. For instance, when several marbles are rolling, each marble will eventually hit a blocked marble, will be blocked by a switch, change a switch state, or disappear when arriving at an *End* piece. For each of these events, one can easily compute the point in time when this event happens as long as no other event happens first, changing the system state. However, since we are interested in the first of these events only when calculating ε and τ, we need not consider situations when events influence other events. Hence,

calculation of the next internal events can be realized as follows: Whenever a new ASS state is reached, i.e., when the graph is changed, a list of all possible internal events for each part of the system is computed. Only that event is selected that will happen first. This implicitly realizes ε and τ. If there is no unique first event, one of them is chosen nondeterministically. When the event is triggered at the scheduled point in time, or if an external event happens earlier, the graph will be modified, i.e., a state transition is performed, and the next internal event will be computed again.

As already described, triggering an external or internal event actually means performing a GT; a partial morphism selects where the GT is applied. For external events, the partial morphism is selected by the external source, e.g., by selecting the *Start* piece where the user places a marble. For an internal event, however, the partial morphism must have been determined when the event has been "scheduled" after the last graph modification. So far, we have not yet discussed how these internal events get calculated. Actually, event (types) are associated with graph patterns together with additional application conditions, i.e., positive and negative application conditions as well as conditions on node and edge attributes as well as the current time. Each pattern describes situations, in which this event type occurs. Therefore, the list of all possible internal events is found by searching for all embeddings of each of these patterns satisfying their application conditions. An additional evaluation rule, called *time calculation rule*, computes the point in time when the event will happen. The results of this rule are used to select the first scheduled internal event. The associated embedding determines where the event takes place. Hence, the left-hand side of the GT rule specifying the effect of the event must consist of the same pattern that specifies the event and whose embedding has already been found when scheduling the event.

In summary, the specification of an internal event type consists of a graph transformation rule with application conditions and a related time calculation rule. An external event type can be specified with a rule only.

We have extended our meta-tool DIAMETA [Min06] such that not only static visual languages can be specified, but also animated visual languages following the ideas described above. DIAMETA allows for generating editing environments for visual animated models from such specifications. The DIAMETA framework is now aware of events, and it manages an event queue that is used to determine the next internal events. This event queue is actually built up and updated in a smart way based on changes of the graph model.

The specification of events in DIAMETA is not restricted to single GT rules. Graph transformation programs, which may consist of several rules, are used instead. The application of these rules is controlled by additional control programs which, e.g., may specify a sequence of rules or use more complex control operators like *apply as long as possible* [Min02]. The availability of complex control programs allows for the specification of arbitrarily powerful GT transformations although each single GT rule is just a simple SPO rule with optional negative application conditions.

5 Specification of Avalanche

This section outlines how visual animated models of *Avalanche* are specified in DIAMETA. One purpose of the described specifications was generating an editing environment which is able to

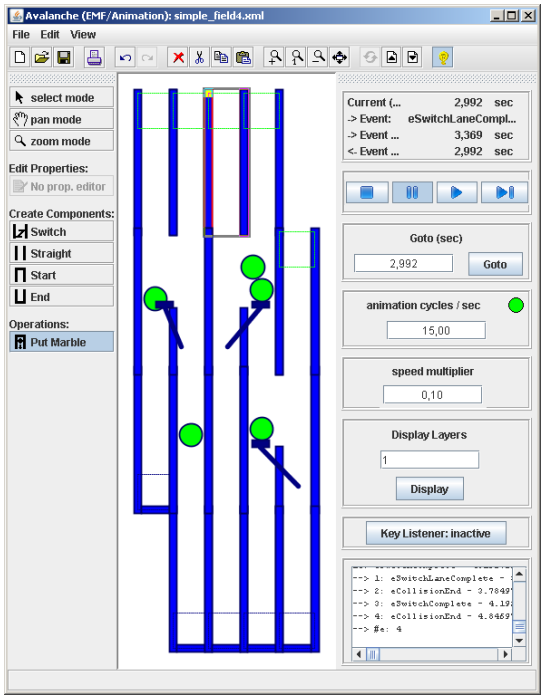


Figure 4: *Avalanche* editor screenshot

(a) build *Avalanche* boards and (b) play the game including the possibility to put marbles onto the *Avalanche* board and watch the progress of the system. A screenshot of the resulting editor is shown in Figure 4. An animated example can be found online[2].

Typed, attributed hypergraphs are used for representing models, i.e., animated diagrams. Each model component is represented by a hyperedge that visits the nodes representing the component's attachment points. Model hypergraphs also contain relation edges (binary hyperedges), that stand for relationships between components, and further hyperedges (called *animation edges* in the following) that are used for the representation of animation states only. More details about the usage of hypergraphs for the specification of visual languages (except hyperedges representing animation states) can be found in [Min06].

The *Avalanche* model components are the ones shown in Figure 3 with the hyperedge types *start*, *end*, *switch*, *straight* and *marble*. Figure 5 shows an example *Avalanche* board and its model hypergraph. Each component is associated with its component hyperedge, which is depicted by a filled rectangle each. Nodes are drawn as small circles. For instance, the *switch* hyperedge is connected to individual nodes via connectors ("tentacles") 0 to 3. The four nodes represent the top-left, top-right, bottom-left and bottom-right corners of the switch. Each of them can be connected to another model component. Connections between model components are represented by binary *at_topbottom* relation hyperedges. They are depicted by fat arrows. Additional edge types are used for animation edges representing the animation state, e.g., edge

[2] http://www.unibw.de/inf2/DiaGen/animated

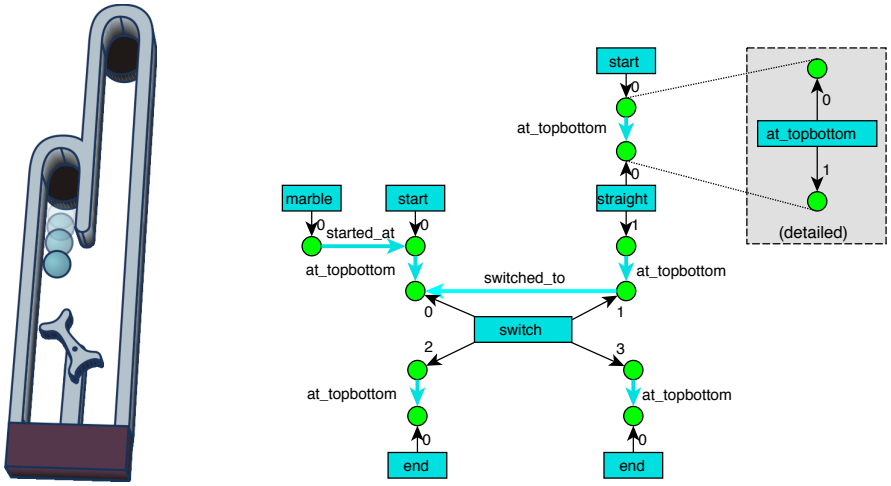


Figure 5: *Avalanche* board and corresponding hypergraph

types *switched_to* or *switching_to*. The *switched_to* edge connects the first switch node with the second one if the right lane is blocked by the switch. If the left lane is blocked, the edge connects these nodes the other way around. Analogously, the *switching_to* edge indicates that the particular lane is not blocked yet, but the switch is currently moving in order to block the lane afterwards. There are also different edge types representing a marble's state as explained later.

Furthermore, each component hyperedge has the layout attributes x and y for the position of the represented diagram component. Finally, hyperedges which can be animated (resp. their corresponding components) contain an attribute *tc*. It is used for storing the point in time when attributes of the hyperedge or its state have been modified lastly.

Static components and their visual appearance can be specified like in static DIAMETA. For animated components, the specification is slightly different. There are two types of animations which are represented by a subgraph: a rolling marble and a shifting switch. Figure 6 shows two concrete graph examples which represent these animations: (a) represents a switch which is currently shifting from left to right. The *switching_to* edge is an animation edge that represents the shifting state of the switch. The shifting animation has started at 20000 ms, which is indicated by attribute *tc*. (b) represents a marble which is rolling. This animation has started at 3000 ms when the marble was at position $(110, 10)$. The *started_at* edge is again an animation edge which specifies the animation state. The edge is not necessary for static diagrams, but it rather indicates the component where the marble was located when the animation started

By using these attributes and animation edges, the animated visual appearance for a component depending on the proceeding time t can be specified. For example, instead of drawing the marble at the static position (x,y), the position for each animation frame is calculated using the time difference $t - tc$ and a linear (or accelerated) movement. Further details about required constants (e.g., rolling speed of the marble, acceleration, relation points for positions) are omitted here.

So far, the described specification is sufficient for editable models with basic animations like rolling marbles or shifting switches. However, interaction between model elements or user in-

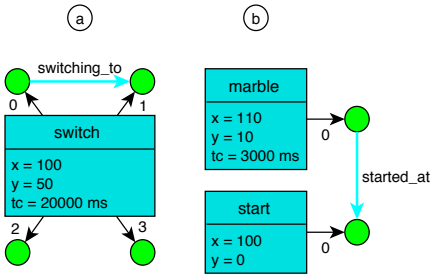


Figure 6: Hyperedges of animated components

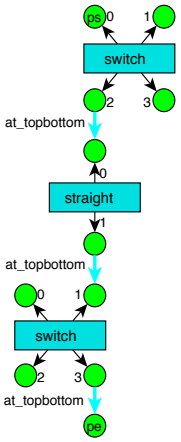


Figure 7: Example path

teraction during animation have not been considered yet. In order to specify the *Avalanche* behavior, the following internal and external events are specified. Please note that some events must be specified for the left and right side of switches separately[3]. Because the specification of both sides is analogous, rules for the right side are omitted.

- External events:
  - *PutMarble*: The user selects a *Start* component in order to place a marble there.
- Internal events:
  - *MarbleStopLeft*: A rolling marble hits the top of a switch (tilted to the left) and is blocked.
  - *ReleaseMarble*: A marble, blocked by a switch, is released and starts rolling down because the switch does not block the lane any more.
  - *MarbleSwitchingLeft*: A rolling marble hits the bottom of a switch (tilted to the right) and initiates the shifting of the switch; during this shifting process, the switch cannot block marbles or be shifted again.
  - *SwitchingCompleteLeft*: A shifting switch reaches its final destination after it has started shifting from the right to the left side.
  - *MarbleChangeLaneLeft*: A rolling marble starts changing the lane because it hits another marble that is blocked by a switch tilted to the left.
  - *ChangeLaneCompleteLeft*: A marble reaches its new lane after it has started changing its lane from the left to the right side.
  - *RemoveMarble*: The marble reaches the end of the lane and is removed from the board.

[3] DIAMETA actually supports more generic specifications that cover both sides, but they are too technical and less suited for presentation in this paper.

In the rest of this section, these events are described. Figure 8 shows the event specifications by GT rules[4] and, for internal events, time calculation rules (indicated by the *Time* keyword). DIAMETA actually uses a different, primarily textual syntax; the syntax in Figure 8 is used for illustration only. It shows the GT rule within one pattern: parts which are removed by the rule are drawn in red and marked with "- - -", and parts which shall be added are drawn in green with "+++". Attribute modifications are illustrated by expressions within a separate *Actions* box. Please note that some expressions make use of constants starting with letter *C*; these constants represent specification details, e.g., rolling speed, relation points for positions, etc.

The external event *PutMarble* is triggered by the user who selects a *start* component and calls an operation called *PutMarble* (by clicking the corresponding button in the *Avalanche* editing environment). The selected *start* component is then used for defining a partial match for the pattern of the graph transformation rule specified for *PutMarble*. The result of the rule is a created marble, which starts rolling down the lane.

SwitchingCompleteLeft is a simple internal event. Events of this type occur for each *switch* edge with a *switching_to* animation edge as shown in the pattern, i.e., for each switch which is currently shifting from the right to the left side. The time of the corresponding event is determined by a simple formula indicating that the switch has reached the final destination after a constant amount *C*5 of time. Hereby, the value in attribute *tc* represents the point in time when the corresponding switch started shifting (triggered by event *MarbleSwitchingLeft*, see below). As a result, the *switching_to* edge is replaced by a *switched_to* edge.

MarbleStopLeft is a more complex event specification because it has to describe a rolling marble hitting a blocking switch in the marble's lane. The lane can go through a number of *start*, *straight*, and *switch* components. In the event specification, this is represented by a dashed arrow indicating a path within the model hypergraph. The path is actually an arbitrary sequence of *at_topbottom(0,1)*, *switch(0,2)*, *switch(1,3)*, or *straight(0,1)* elements. Thereby, the numbers in parenthesis specify the hyperedge tentacles the path must follow: the first number specifies the ingoing tentacle and the second one specifies the outgoing tentacle when following the path through hyperedges. An example path is shown in Figure 7: the path starts at node *ps* and ends at node *pe*. In a matching scenario of *MarbleStopLeft*, the *marble* edge must be linked to *ps* via the *started_at* edge, and *pe* is the node of the first outgoing tentacle of *switch* edge *sw*.

The time of the event *MarbleStopLeft* is calculated based on the distance between the rolling marble and the switch. As a result of this event, the marble looses its *started_at* relation and receives a *blocked* edge instead, which indicates the marble being blocked by the switch. Moreover, the marble's static position is set to the position of the switch.

Event *MarbleSwitchingLeft* is similar and also makes use of path expression *path_down* described above. The switch, however, does not block the rolling marble's lane, indicated by the *switched_to* edge having the opposite direction of the one in event *MarbleStopLeft*. If this event occurs, the marble continues rolling normally, but it also shifts the switch. The switch state is changed by replacing the *switched_to* edge by a *switching_to* edge, now being associated with the opposite side/lane. Nevertheless, the switch is not considered blocking this lane yet. The end of this shifting phase will trigger a new *SwitchingCompleteLeft* event (see above).

[4] Please note that these events are actually specified using single GT rules; the specification of *Avalanche* does not require complex graph transformation programs as event specifications.

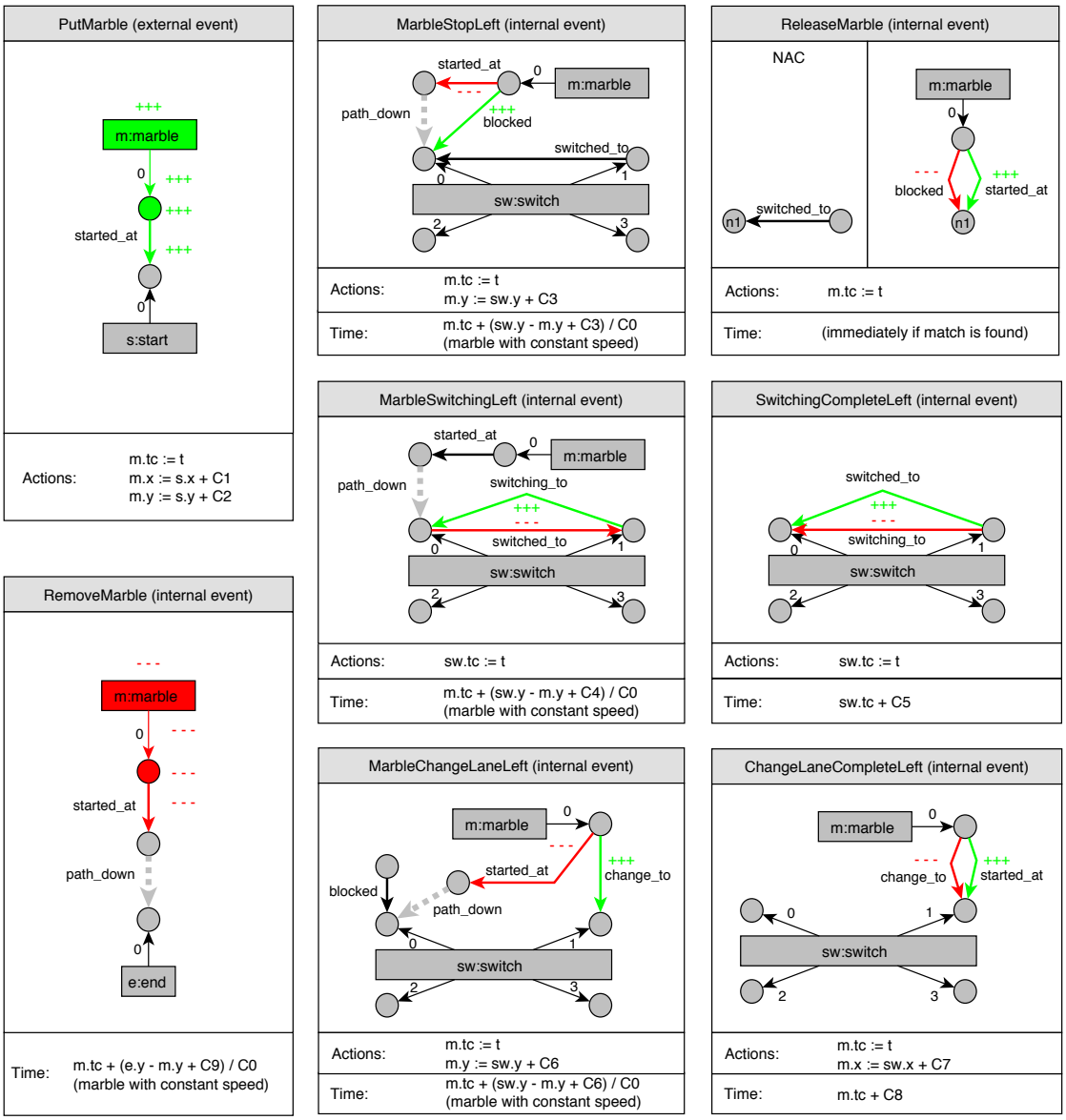


Figure 8: *Avalanche* Event Specifications

The event *ReleaseMarble* can occur directly after a *MarbleSwitchingLeft* (or *MarbleSwitchingRight*, resp.) event. The *ReleaseMarble* event does not specify a time calculation rule and, therefore, is applied immediately as soon as a match is found. It releases a blocked marble if the switch does not block the according lane any more. The marble then starts rolling down again, indicated by the animation edge *started_at*.

The internal event *MarbleChangeLaneLeft* handles the case that a rolling marble hits a blocked marble. The rolling marble then changes its lane, which is indicated by a *change_to* edge linked to the switch node of the opposite side. This changing process ends as soon as event *Change-*

LaneCompleteLeft occurs after a constant amount of time. The marble then continues rolling down again, however in the switch's other lane.

Finally, event *RemoveMarble* occurs as soon as a marble reaches the end of its lane. The marble and the associated *started_at* edge are removed then.

6 Related Work

An approach of simulating and animating visual languages has already been described in [Erm06]. However, the described methods come along with some of the issues mentioned in the introduction of this paper. The resulting animations are self-running movies, and amalgamated graph transformation rules are already required for the specification of less complex examples like animated petri nets, for instance.

The described abstract animation system is similar to timed event systems and, therefore, also to *DEVS* [ZPK00]. However, abstract animation systems omit some specific features like acceptance stated (compared to timed event systems in general) or output function (compared to *DEVS*). On the other hand, the intention of abstract animation systems is that system states can be visualized in an animated way, and state changes start, stop, and change these animations. It allows for an easier specification of systems which must be animated and illustrated.

Section 5 shows that there is a need for a time attribute like *tc* within graph vertices for many types of animation. This attribute can be compared with an attribute *chronos* introduced in [GHV02] which describes an approach of graph transformation with time. The shown transformations utilize logical clocks, which are also represented by the mentioned vertex attribute.

Another, but similar approach, though not based on GTs, is shown in [EVV09]. This work describes a transformation system enriched by parameters like duration, repetitions and focuses on the visual notation of such rules.

The idea of states describing animation and behavior of graphical objects is a common approach. The term *animation state* has also been described in [Vit05] where it represents a state in which corresponding object attributes are changing with regard to animations. The work also shows how animations and the behavior of object can be described by a visual language based on UML2 statecharts.

7 Conclusions

The specification of animated visual languages based on graphs has been limited with regard to simultaneous, independent animations and interactivity yet. The described approach enables dynamic, animated and interactive models using existing graph and graph transformation techniques. After minor extensions, the existing meta-tool DIAMETA is able to generate editing environments for interactive animated visual models like *Avalanche*.

However, the way from a complex dynamic system to an event-oriented and graph-based system still is a challenging task. Right now, we are investigating additional techniques in order to specify dynamic systems on a higher level, which is less formal and also diagrammatic. Resulting diagrams shall help deriving graph-based specifications and visualizations then. Also tool support for some of the described methods still lacks usability, e.g., visualization and time cal-

culation rules must be written by hand. Animation patterns for common animation types would be desirable, and also a physics engine or other frameworks could be used in order to simplify specification of particular kinds of animated visual languages.

Bibliography

[Erm06] C. Ermel. *Simulation and Animation of Visual Languages based on Typed Algebraic Graph Transformation*. PhD thesis, Tech. Univ. Berlin, Fak. IV, Books on Demand, Norderstedt, 2006.

[EVV09] J. Eduardo Rivera, C. Vicente-Chicote, A. Vallecillo. Extending visual modeling languages with timed behavioral specifications. In *IDEAS 2009: Proc. 12th Iberoamerican Conf. on Requirements Engineering and Software Environments*. Pp. 87–100. Colombia, April 2009.

[GHV02] S. Gyapay, R. Heckel, D. Varró. Graph Transformation with Time: Causality and Logical Clocks. In *ICGT '02: Proc. 1st Int. Conf. on Graph Transformation*. Pp. 120–134. Springer-Verlag, 2002.

[LV02] J. d. Lara, H. Vangheluwe. AToM3: A Tool for Multi-formalism and Meta-modelling. In *FASE '02: Proc. 5th Int. Conf. on Fundamental Approaches to Software Engineering*. Pp. 174–188. Springer-Verlag, London, UK, 2002.

[Min02] M. Minas. Concepts and Realization of a Diagram Editor Generator Based on Hypergraph Transformation. *Science of Computer Programming* 44(2):157–180, 2002.

[Min06] M. Minas. Generating Meta-Model-Based Freehand Editors. In *Proc. of the 3rd Int. Workshop on Graph Based Tools (GraBaTs'06), Satellite of ICGT'06*. Electronic Communications of the EASST 1. 2006.

[SM09] T. Strobl, M. Minas. Implementing an Animated Lambda-Calculus. In *Workshop on Visual Languages and Logic, Satellite of VL/HCC'09*. CEUR Workshop Proceedings 510. 2009.

[Smi86] R. B. Smith. The Alternate Reality Kit: An Animated Environment for Creating Interactive Simulations. In *Proc. of the 1986 IEEE Computer Society Workshop on Visual Languages*. Pp. 99–106. 1986.

[Vit05] A. Vitzthum. SSIML/Behaviour: Designing Behaviour and Animation of Graphical Objects in Virtual Reality and Multimedia Applications. In *Proc. Seventh IEEE Int. Symp. on Multimedia (ISM'05)*. Pp. 159–167.

[ZPK00] B. Zeigler, H. Praehofer, T. G. Kim. *Theory of Modeling and Simulation*. Academic Press, January 2000.

Electronic Communications of the EASST
Volume of the Pre-proceedings of GT-VMT (2010)

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

Decidable Race Condition and Open Coregions in HMSC

Vojtěch Řehák Petr Slovák Jan Strejček Loïc Hélouët

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

Decidable Race Condition and Open Coregions in HMSC

Vojtěch Řehák[1*] **Petr Slovák**[1†] **Jan Strejček**[1‡] **Loïc Hélouët**[2]

[1]Faculty of Informatics, Masaryk University, Brno, Czech Republic
[2]INRIA/IRISA, Rennes, France

Abstract: *Message Sequence Charts (MSCs)* is a visual formalism for the description of communication behaviour of distributed systems. An MSC specifies relations between communication events with partial orders. A situation when two visually ordered events may occur in any order during an execution of an MSC is called a *race* and is usually considered as a design error. While there is a quadratic time algorithm detecting races in a finite communication behaviours called *Basic Message Sequence Charts (BMSCs)*, the race detection problem is undecidable for *High-level Message Sequence Charts (HMSCs)*, an MSC formalism describing potentially infinite sets of potentially unbounded behaviours. To improve this negative situation for HMSCs, we introduce two new notions: a new concept of race called trace-race and an extension of the HMSC formalism with open coregions, i.e. coregions that can extend over more than one BMSC. We present three arguments showing benefits of our notions over the standard notions of race and HMSC. First, every trace-race-free HMSC is also race-free. Second, every race-free HMSC can be equivalently expressed as a trace-race-free HMSC with open coregions. Last, the trace-race detection problem for HMSC with open coregions is decidable and PSPACE-complete. Finally, the proposed extension of coregions allows to represent in a visual fashion whether an arbitrary number of racing events in the usual MSC formalism are concurrent or not.

Keywords: HMSC; race condition; trace-race condition; open coregions;

1 Introduction

Message Sequence Chart (MSC) [ITU04] is a popular visual formalism for specification of distributed systems behaviours (e.g. communication protocols or multi-process systems). Its simplicity and intuitiveness come from the fact that MSC describes only exchange of messages between system components, while other aspects of the system (e.g. content of the messages, computation steps) are abstracted away. Even such an incomplete model can indicate serious errors in the designed system. This paper focuses on a common error called *race condition*.

MSCs are based on composition of simple chronograms called *Basic Message Sequence Charts (BMSCs)*. A BMSC consists of a finite number of *processes* and *events*. Processes are represented by vertical lines, and all events executed by some process are located on its lifeline

* Partially supported by Czech Science Foundation (GAČR), grants No. 201/08/P459 and No. P202/10/1469.
† Partially supported by the research centre "Institute for Theoretical Computer Science (ITI)", project No. 1M0545.
‡ Partially supported by Czech Science Foundation (GAČR), grant No. 201/08/P375, and Ministry of Education of the Czech Republic, project No. MSM0021622419.

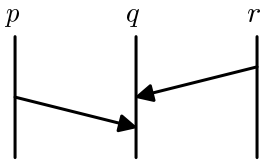


Figure 1: A BMSC containing a race

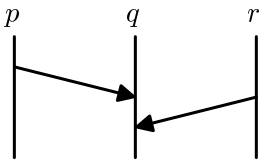


Figure 3: A similar BMSC containing a race

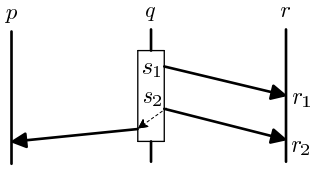


Figure 5: A BMSC containing a race between r_1, r_2

and ordered from top to bottom. Messages are represented by an arrow from a sending event to a receiving event. Total orderings of events on lifelines and messages form a *visual order* $<$, which provides graphically information on the respective ordering of events. However, the visual order can not always be enforced by the architecture of the modelled system. In addition to the visual order, there exists a *causal order* $\ll$, that is weaker than $<$. Intuitively, events e, f are in causal order $e \ll f$, if the BMSC enforces that e always precedes f. There are several definitions of causal order depending on the settings of the modelled system and semantics of the model. For example, if one process sends two messages to another process, the corresponding receive events are causally ordered if and only if the considered message transport protocol has the *FIFO property*: two messages sent from one process to another are always received in the same order. In this paper, we assume that every process has one unbounded buffer for all incoming messages and that the message transport protocol satisfies the FIFO property.

A BMSC contains a *race condition* (or simply *race*) [AHP96] if there are two visually ordered events that are not causally ordered (i.e. they can actually occur in an arbitrary order). For example, Figure 1 depicts that the process q receives a message from r followed by a message from p. As processes and communication in BMSCs are always asynchronous, the messages can be also received in the opposite order as shown in Figure 2. In both figures, the two receive events are in race as they are ordered visually but not causally. Races in BMSC description should be considered as a design error, as they exhibit discrepancies between the intended ordering designed in a BMSC, and the ordering that a real implementation of this BMSC would enforce. Races in a BMSC can be detected in quadratic time [AHP96].

While a BMSC describes only a single and finite communication scenario, its extension called *High-level Message Sequence Chart (HMSC)* [RGG96, AHP96] can describe more complex interactions, with iterations and alternatives between several scenarios. An HMSC is a finite state transition system where each state is labelled by a BMSC or a reference to another HMSC. In the sequel, we will only consider HMSCs labelled by BMSCs. Each run (i.e. a path starting in the initial state and ending in a final state) of an HMSC can be understood as a single BMSC, which is a concatenation of the BMSCs labelling the states along the run. Hence, an HMSC represents a potentially infinite set of BMSCs of unbounded size.

The definition of race was extended to HMSCs in [MP99]. Roughly speaking, an HMSC H has a race if some BMSC represented by H contains a race and H does not represent any BMSC where the two racing events are defined with the opposite visual order. Unfortunately, the problem whether a given HMSC contains a race is undecidable [MP99, ITU04].

In this paper, we propose an alternative definition of race for HMSCs called *trace-race*. Intuitively, an HMSC has a trace-race if some BMSC represented by H contains a race. Clearly,

every trace-race-free HMSC is also race-free but not vice versa. To improve the expressive power of trace-race-free HMSCs, we extend the HMSC formalism with *open coregions*. A coregion is a standard part of the MSC formalism that allows some events on the same process in a BMSC to be visually unordered. In particular, coregions can be used to visually order only causally related events (hence making concurrency a visual property). While this application of coregions can remove all races in BMSCs, it is not sufficient for removing all races in HMSCs. An open coregion is basically a coregion spread over several BMSCs. We present a transformation of an arbitrary race-free HMSC into an equivalent trace-race-free HMSC with open coregions, where equivalence means that the two HMSCs have the same linearizations. Finally, we show that the problem whether a given HMSC with open coregions contains a trace-race is decidable and PSPACE-complete. In fact, our algorithm is polynomial for HMSCs with fixed number of processes and gates. For definitions of gates and linearizations see Sections 2 and 3, respectively.

The rest of the paper is organized as follows. Section 2 recalls the definitions of BMSCs, HMSCs, and race condition for BMSCs. The race and trace-race conditions for HMSCs are defined and compared in Section 3. Section 4 is devoted to the translation of race-free HMSCs into equivalent trace-race-free HMSCs. The decidability and complexity of the trace-race detection problem is discussed in Section 5. Section 6 briefly summarizes benefits of the presented notions. Due to the space limitations, we present only crucial lemmata and theorems accompanied by explanations of basic ideas. Proofs with all technical details can be found in [ŘSSH09].

2 Preliminaries

The following definitions omit some features of MSCs given by the ITU standard [ITU04], e.g. atomic actions, labelling of messages with names, timers etc. However, these restrictions are quite common, and our results can be extended to MSCs with atomic actions and message labelling using the technique of [DGH08].

2.1 BMSCs with (open) coregions, gates, and general ordering

The basic concepts of BMSCs are described in Section 1. In the visual representation of a BMSC, processes are depicted as vertical lines and messages are represented by arrows between these lines. Events located on the same process line are visually ordered from top to bottom. A process line may contain segments called *coregions* delimiting subsets of events. Events in a coregion are a priori not in visual order, but they can be visually ordered using a *general ordering* relation. (this relation need not be a partial order). Coregions are visually represented by rectangles and general ordering by dashed arrows between pairs of ordered events (see Figure 3).

In existing MSC formalisms, coregions are limited to finite set of events located in a single BMSC. We extend the definition of BMSCs with *open coregions* and *gates*. These features allow coregions of arbitrary size, spread over several concatenated BMSCs. Gates enable events of different BMSCs to be generally ordered within the final joined coregion. Similar ideas for connecting orders using gates or predicates was already proposed for instance in [Pra86, GH07].

A coregion can be open on top (*top-open coregion*), on bottom (*bottom-open coregion*), or open on both sides. All processes use a common gate name space $\mathbf{G}$. For each process p, we

define the sets of *top gates* $p.\overline{\mathbf{G}} = \{p.\overline{g} \mid g \in \mathbf{G}\}$ and *bottom gates* $p.\underline{\mathbf{G}} = \{p.\underline{g} \mid g \in \mathbf{G}\}$ located on process p. Given a BMSC with a set of processes $\mathbf{P}$, we set $\mathbf{P}.\overline{\mathbf{G}} = \bigcup_{p\in\mathbf{P}} p.\overline{\mathbf{G}}$ and $\mathbf{P}.\underline{\mathbf{G}} = \bigcup_{p\in\mathbf{P}} p.\underline{\mathbf{G}}$ to be the sets of all top and bottom gates in this BMSC, respectively. We also extend the general ordering to range over both events and gates within an open coregion.

Definition 1 Let $\mathbf{G}$ be a finite gate name space. A BMSC over $\mathbf{G}$ is a tuple $M = (\mathbf{P}, E_S, E_R, P, \{<_p\}_{p\in\mathbf{P}}, \mathbf{M}, \mathbf{C}, \{\prec_C\}_{C\in\mathbf{C}})$ where

- $\mathbf{P}$ is a finite set of processes.
- E_S, E_R are disjoint finite sets of *send* and *receive events*, respectively. We set $E = E_S \cup E_R$.
- $P : E \to \mathbf{P}$ is a mapping that associates each event with a process.
- $<_p$ is a total order on all events on a process p.
- $\mathbf{M} \subseteq (E_S \times E_R)$ is a bijective mapping, relating every send with a unique receive. We assume that a process cannot send a message to itself, i.e. $P(e) \neq P(f)$ whenever $(e,f) \in \mathbf{M}$. For any $(e,f) \in \mathbf{M}$, we use $\mathbf{M}(e)$ to denote the receive event f, and $\mathbf{M}^{-1}(f)$ to denote the send event e.
- $\mathbf{C}$ is a finite set of pairwise disjoint *coregions*, where a coregion $C \in \mathbf{C}$ is a consistent nonempty subset of events and gates of a single process p, i.e.
  - $\emptyset \neq C \subseteq P^{-1}(p) \cup p.\overline{\mathbf{G}} \cup p.\underline{\mathbf{G}}$ for some $p \in \mathbf{P}$
  - if $e <_p d <_p f$ and $e, f \in C$, then $d \in C$.

  A coregion C containing a top gate is called *top-open* and it has to contain all top gates $p.\overline{\mathbf{G}}$ and satisfy that if $e <_p f$ and $f \in C$ then $e \in C$. A coregion C containing a bottom gate is called *bottom-open* and it has to contain all bottom gates $p.\underline{\mathbf{G}}$ and satisfy that if $e <_p f$ and $e \in C$ then $f \in C$. A coregion which is both top-open and bottom-open is called just *open*.
- $\prec_C$ is an acyclic relation called *general ordering* on elements in C such that $\prec_C \subseteq (p.\overline{\mathbf{G}} \cup P^{-1}(p)) \times (p.\underline{\mathbf{G}} \cup P^{-1}(p))$, where p is the process containing the coregion C.

The definition says that a top-open coregion has to contain all top gates. As coregions are pairwise disjoint, there is at most one top-open coregion. Similarly, each BMSC contains at most one bottom-open coregion. Note that we do not impose that coregions contain events. For example, an open coregion covering an inactive process can connect top and bottom gates.

In the visual representation, an open coregion is depicted as a rectangle without the side(s) which are open. Gates are represented by small squares on the corresponding missing side of these rectangles. As gates are always depicted in the same order, their names become redundant (and they are often omitted). For example of BMSCs with open coregions see Figure 4. Recall that dashed arrows represent a general ordering.

2.2 Visual order, causal order and race in BMSCs

Every BMSC induces two preorders on events: visual $<$ and causal $\ll$ ordering. The *visual order* represents the order of events directly described by the BMSC. Loosely speaking, $<$ is the reflexive and transitive closure of total orders $<_p$ of events on each process, excluding the order of events within each coregion, plus general ordering and the order generated by the FIFO

property and the fact that every send event precedes the corresponding receive event. The visual order is actually defined over the union of events and gates.

Definition 2 Let $M = (\mathbf{P}, E_S, E_R, P, \{<_p\}_{p\in\mathbf{P}}, \mathbf{M}, \mathbf{C}, \{\prec_C\}_{C\in\mathbf{C}})$ be a BMSC over $\mathbf{G}$. A *visual order* $<$ given by M is the least preorder $< \subseteq (\mathbf{P}.\overline{\mathbf{G}} \cup E) \times (\mathbf{P}.\underline{\mathbf{G}} \cup E)$ such that

(i) $<$ contains the relation $\big(\bigcup_{p\in\mathbf{P}} <_p \;\setminus\; \bigcup_{C\in\mathbf{C}} C\times C\big) \;\cup\; \big(\bigcup_{C\in\mathbf{C}} \prec_C\big) \;\cup\; \mathbf{M}$,

(ii) $<$ respects the FIFO property, i.e. for every $e, f \in E_S$ such that $P(e) = P(f)$ and $P(\mathbf{M}(e)) = P(\mathbf{M}(f))$, it holds that $e < f$ implies $\mathbf{M}(e) < \mathbf{M}(f)$.[1]

One can define a BMSC where $<$ is not a partial order. This situation is clearly a design error and it can be detected by a cycle detection algorithm. In the sequel, we always assume that $<$ is a partial order.

In contrast to the visual order, the *causal order* captures the partial order of events that has to be respected by all executions as it is enforced by the semantics of the design. Hence, the causal order represents the interpretation of a BMSC relevant to its implementation.

Definition 3 Given a BMSC $M = (\mathbf{P}, E_S, E_R, P, \{<_p\}_{p\in\mathbf{P}}, \mathbf{M}, \mathbf{C}, \{\prec_C\}_{C\in\mathbf{C}})$ over $\mathbf{G}$, we define a *causal order* $\ll$ as the least partial order on E such that $e \ll f$, if

- $(e, f) \in \mathscr{M}$, i.e. *send and receive events of each message are ordered*, or
- $P(e) = P(f)$ and $e < f$ and $f \in E_S$, i.e. *any send event is delayed until all previous events took place*, or
- $P(e) = P(f)$ and $\exists e', f' \in E$ such that $e' < f'$, $P(e') = P(f')$, $(e', e) \in \mathscr{M}$ and $(f', f) \in \mathscr{M}$, i.e. *causal order respects the FIFO property*.

Lemma 1 *For every BMSC it holds $\ll \subseteq <$. Further, for each $f \in E_S$ it holds $e < f \iff e \ll f$.*

A race is defined as a difference between visual and causal order on events.

Definition 4 If a BMSC contains some events e, f satisfying $e < f$ and $e \not\ll f$, we say that the BMSC contains a *race (between events e, f)*. Otherwise, the BMSC is called *race-free*.

Theorem 1 ([AHP96]) *The problem whether a given BMSC with n events contains a race is decidable in time $\mathcal{O}(n^2)$.*

Note that [AHP96] deals with BMSCs without any coregions. However, an extension of this theorem to BMSCs with (possibly open) coregions and general ordering is straightforward.

The following lemma says that a BMSC contains a race if and only if it contains a race between two events on the same process.

Lemma 2 *A BMSC contains a race if and only if there are two events e, f such that $e < f$, $e \not\ll f$, and $P(e) = P(f)$.*

[1] The FIFO property is usually not included in the definition of a visual order. However, once we choose the FIFO message passing setting, violating this property should be considered as a design error and it can be easily detected. Hence, we included the property directly in our definition.

2.3 HMSCs

An HMSC is a finite directed graph with an *initial* state and a set of *final* states, where each state is labelled with a BMSC.

Definition 5 An *HMSC* is a tuple $(S, \rightarrow, s_0, S_F, L, \mathscr{L})$ where

- S is a finite set of states, $s_0 \in S$ is an *initial state*, $S_F \subseteq S$ is a set of *final states*,
- $\rightarrow \subseteq S \times S$ is a transition relation,
- $\mathscr{L}$ is a finite set of BMSCs over a common gate name space,
- $L(s) : S \rightarrow \mathscr{L}$ is a mapping assigning to each state a BMSC.

A sequence of states $\sigma = s_1 s_2 \cdots s_k$ is a *path*, if $(s_i, s_{i+1}) \in \rightarrow$ for every $1 \leq i < k$. A path is a *run* if $s_1 = s_0$ and $s_k \in S_F$.

To give a semantics of HMSCs, we need to define a concatenation operation on BMSCs. Intuitively, the concatenation of BMSCs M_1 and M_2 is done by gluing the corresponding process lines together with the BMSC M_2 drawn beneath M_1. If M_1 and M_2 contain bottom-open and top-open coregions on a process p, respectively, then the two coregions are merged and each bottom gate $p.\underline{g}$ of the upper open coregion is joined with the corresponding top gate $p.\overline{g}$ of the lower open coregion. Further, whenever an event e of the upper coregion is generally ordered with a joined gate and this joined gate is generally ordered with an event f of the lower coregion, the events e, f become generally ordered in the newly created coregion. The joined gates are then removed. If M_1 contains a bottom-open coregions on a process p that is not in M_2, then the coregion remains bottom-open. However, if M_1 contains a bottom-open coregion on a process p and M_2 contains the process p without any top-open coregion on it, then the bottom side of the coregion is closed.

Definition 6 Let $M_i = (\mathbf{P}_i, E_{Si}, E_{Ri}, P_i, \{<_{ip}\}_{p \in \mathbf{P}}, \mathbf{M}_i, \mathbf{C}_i, \{\prec_{iC}\}_{C \in \mathbf{C}_i})$ for $i = 1, 2$ be two BMSCs over a common gate name space $\mathbf{G}$ and such that the sets $E_{S1} \cup E_{R1}$ and $E_{S2} \cup E_{R2}$ are disjoint (we can always rename events so that the sets become disjoint). The concatenation of M_1 and M_2 is the BMSC $M_1 \cdot M_2 = (\mathbf{P}_1 \cup \mathbf{P}_2, E_{S1} \cup E_{S2}, E_{R1} \cup E_{R2}, P_1 \cup P_2, \{<_p\}_{p \in \mathbf{P}}, \mathbf{M}_1 \cup \mathbf{M}_2, \mathbf{C}, \{\prec_C\}_{C \in \mathbf{C}})$ where

$$<_p = \begin{cases} <_{1p} \text{ if } p \in \mathbf{P}_1 \setminus \mathbf{P}_2 \\ <_{2p} \text{ if } p \in \mathbf{P}_2 \setminus \mathbf{P}_1 \\ \text{transitive closure of } <_{1p} \cup <_{2p} \cup (P_1^{-1}(p) \times P_2^{-1}(p)) \\ \text{if } p \in \mathbf{P}_1 \cap \mathbf{P}_2 \end{cases}$$

and $\mathbf{C}$ contains all coregions C of the following five kinds:

1. $C \in \mathbf{C}_1$ and C is not bottom-open or C is on a process $p \in \mathbf{P}_1 \setminus \mathbf{P}_2$. We set $\prec_C = \prec_{1C}$.
2. $C \in \mathbf{C}_2$ and C is not top-open or C is on a process $p \in \mathbf{P}_2 \setminus \mathbf{P}_1$. We set $\prec_C = \prec_{2C}$.
3. $C = C_1 \setminus p.\underline{\mathbf{G}}$ for some $C_1 \in \mathbf{C}_1$ such that $p.\underline{\mathbf{G}} \subseteq C_1$ and $p.\overline{\mathbf{G}} \cap C_2 = \emptyset$ for all $C_2 \in \mathbf{C}_2$, i.e. C corresponds to a coregion of $\mathbf{C}_1$ that is bottom-open but there is no matching top-open coregion in $\mathbf{C}_2$ (note that C is closed on the bottom side). We set $\prec_C = \prec_{1C_1} \cap (C \times C)$.

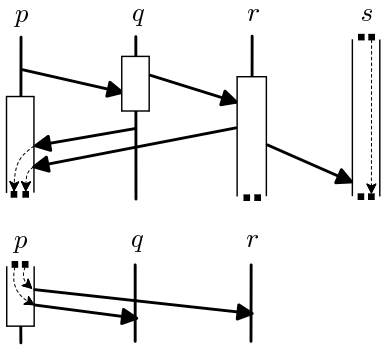


Figure 6: BMSCs M_1 (upper) and M_2

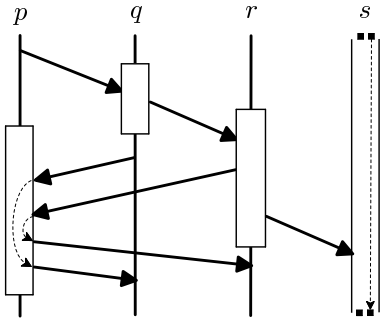


Figure 8: Concatenation $M_1 \cdot M_2$

4. $C = C_2 \setminus p.\overline{\mathbf{G}}$ for some $C_2 \in \mathbf{C}_2$ such that $p.\overline{\mathbf{G}} \subseteq C_2$ and $p.\underline{\mathbf{G}} \cap C_1 = \emptyset$ for all $C_1 \in \mathbf{C}_1$, i.e. C corresponds to a coregion of $\mathbf{C}_2$ that is top-open but there is no matching bottom-open coregion in $\mathbf{C}_1$. We set $\prec_C = \prec_{2C_2} \cap (C \times C)$.
5. $C = (C_1 \setminus p.\underline{\mathbf{G}}) \cup (C_2 \setminus p.\overline{\mathbf{G}})$ for some $C_1 \in \mathbf{C}_1$ and $C_2 \in \mathbf{C}_2$ satisfying $p.\underline{\mathbf{G}} \subseteq C_1$ and $p.\overline{\mathbf{G}} \subseteq C_2$, i.e. C is a bottom-open coregion of $\mathbf{C}_1$ merged with the matching top-open coregion of $\mathbf{C}_2$. We set

$$\prec_C = \{(e,f) \mid \big((e,f) \in \prec_{1C_1} \text{ and } f \notin p.\underline{\mathbf{G}}\big) \text{ or } \big((e,f) \in \prec_{2C_2} \text{ and } e \notin p.\overline{\mathbf{G}}\big) \\ \text{or } \big((e,p.\underline{g}) \in \prec_{1C_1} \text{ and } (p.\overline{g},f) \in \prec_{2C_2} \text{ for some } g \in \mathbf{G}\big)\}.$$

Note that if visual orders of M_1 and M_2 are partial orders, then the visual order of $M_1 \cdot M_2$ is also a partial order. Figures 4 and 5 provide an example of two BMSCs and their concatenation.

Each path $s_1 s_2 \cdots s_k$ of an HMSC represents a single BMSC given by concatenation of the BMSCs assigned to $s_1, s_2, \ldots, s_k$, i.e. a path $\sigma = s_1 s_2 \cdots s_k$ represents the BMSC $L(\sigma) = L(s_1) \cdot L(s_2) \cdot \ldots \cdot L(s_k)$. Hence, an HMSC represents a set of BMSCs corresponding to its runs. As an HMSC may contain a cycle, the represented set of BMSCs can be infinite and there is no bound on the size (i.e. number of events) of such BMSCs.

3 Race conditions in HMSCs

First we explain the idea of race conditions for a set of BMSCs. Let us consider a system where two processes p and r send a message to a third process q, that receives them in arbitrary order. This behaviour can be specified (even without any coregion) by two BMSCs depicted in Figures 1 and 2. Even if both BMSCs contain a race, the specification given by this pair of BMSCs should be considered as race-free because both permutations of the two receive events on process q allowed by causal ordering are included in the specification.

The race condition for a set of BMSCs can formulated very simply using the following terminology. An *execution induced by a BMSC M* is a totally ordered set $(E, \subset)$, where E is the set of events of M and $\subset$ is a linear extension of the causal order $\ll$ given by M. We say that such an execution $(E, \subset)$ *corresponds to a BMSC* M' if M' has the same set of events and $\subset$ is a linear extension of the visual order $<$ of M'.

Definition 7 We say that a set of BMSCs contains a *race* if there exists an execution induced by some BMSC of the set and not corresponding to any BMSC of the set.

The race condition for HMSCs introduced in [MP99] follows the same principle. Unfortunately, we cannot directly say that an HMSC contains a race if it represents a set of BMSCs containing a race. The problem is that the BMSCs represented by the HMSC are constructed with the concatenation operation during which events can be renamed. Therefore, the events are replaced by *labels* keeping the information about sending and receiving processes. Further, the linearly ordered executions are replaced by words called *linearizations*.

Definition 8 Let $M = (\mathbf{P}, E_S, E_R, P, \{<_p\}_{p\in\mathbf{P}}, \mathbf{M}, \mathbf{C}, \{\prec_C\}_{C\in\mathbf{C}})$ be a BMSC. We define an auxiliary function $label : E \to \{p!q, p?q \mid p,q \in \mathbf{P}\}$ such that

$$label(e) = \begin{cases} p!q & \text{if } e \in E_S,\ p = P(e), \text{ and } q = P(\mathbf{M}(e)) \\ p?q & \text{if } e \in E_R,\ p = P(e), \text{ and } q = P(\mathbf{M}^{-1}(e)). \end{cases}$$

A *linearization* of M w.r.t. a partial order $\sqsubset \in \{<, \ll\}$ is a word $label(e_1)label(e_2)\cdots label(e_n)$ such that $E = \{e_1, e_2, \cdots, e_n\}$ and $e_i \sqsubset e_j$ implies $i < j$. Moreover, we define $Lin_\sqsubset(M)$ to be the set of all linearizations of M w.r.t. $\sqsubset$. Finally, we define *linearizations* of an HMSC H w.r.t. $\sqsubset \in \{<, \ll\}$ to be the set

$$Lin_\sqsubset(H) = \bigcup_{\sigma \text{ is a run of } H} Lin_\sqsubset(L(\sigma)).$$

Intuitively, $Lin_\ll(M)$ represents all executions induced by M, while $Lin_<(M)$ represents all executions corresponding to M.

Definition 9 ([MP99]) An HMSC H contains a *race* if $Lin_<(H) \neq Lin_\ll(H)$.

This definition of race has several drawbacks. First of all, the problem whether an HMSC contains a race is undecidable even if we restrict the problem to HMSCs without coregions [MP99, MP00]. Further, as soon as we consider HMSCs with coregions, this notion of race does not tally with the definition of race for BMSCs. For example, the BMSC drawn in Figure 3 contains a race as the messages from q to r can be sent in arbitrary order while the receive events r_1, r_2 are visually ordered. If we look at this BMSC as an HMSC with only one state, then there is no race with respect to Definition 9 as both events r_1, r_2 are represented in linearizations by the same label $r?q$ and therefore the information about their order is lost.

A simple definition of a trace-race follows.

Definition 10 An HMSC H contains a *trace-race* if there is a run σ of H such that the BMSC $L(\sigma)$ contains a race.

If an HMSC H contains no trace-race, then each of its runs σ represents a race-free BMSC $L(\sigma)$. As visual and causal orders of a race-free BMSC coincide, we get that $Lin_<(L(\sigma)) = Lin_\ll(L(\sigma))$. Hence, every trace-race-free HMSC is also race-free. The inverse implication does not hold. For example, the race-free HMSC of Figure 6 has a trace-race (the HMSC describes the system discussed at the beginning of this section).

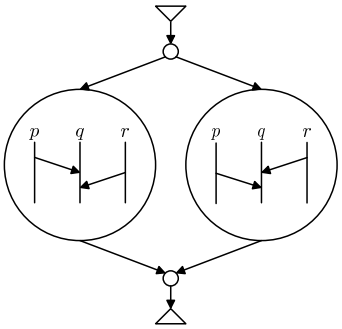


Figure 9: An race-free HMSC with a trace-race

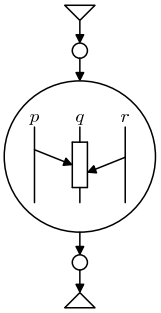


Figure 11: A trace-race-free HMSC

As the definition of trace-race does not replace events by labels, trace-race tallies with the definition of race for BMSCs, i.e. a BMSC has a race if and only if it has a trace-race when seen as a single state HMSC.

It is commonly agreed that designers should avoid races in HMSCs. We provide an intuitive explanation why we think that designers should actually avoid trace-races as well. Let H be a race-free HMSC with a trace-race. As H has a trace-race, there has to be a run σ such that $L(\sigma)$ induces an execution not corresponding to $L(\sigma)$. As H is race-free, this execution corresponds to some BMSC $L(\sigma')$, where $\sigma' \neq \sigma$ is another run of H.[2] Lemma 1 implies that all executions corresponding to a run are also induced by the run. Hence, the execution is in fact induced by (at least) two different runs of the HMSC. This is a potential source of errors as an implementation of this kind of description tends to violate the "write things once" programming principle. Moreover, trace-race-free HMSCs are usually more compact and their use may encourage a cleaner way for designing systems. For example, compare the trace-race-free system depicted on Figure 7, which models the same behaviour as the race-free HMSC of Figure 6.

4 Transformation of HMSCs into trace-race-free HMSCs

We present a transformation of an arbitrary HMSC H into a trace-race-free HMSC H'. The transformation modifies only BMSCs in the states of H. The modified BMSCs have the same processes, events, and causal orders as the original BMSCs, but they induce different visual orders. As the structure of H' remains the same, it has the same set of runs as H. The HMSC is changed in such a way that both visual and causal orders of the BMSC corresponding to a run σ in H' are the same as the causal order of the BMSC corresponding to σ in H. Hence, $Lin_<(H') = Lin_\ll(H') = Lin_\ll(H)$ and H' is trace-race-free. Moreover, if H was race-free (i.e. $Lin_<(H) = Lin_\ll(H)$), then $Lin_<(H) = Lin_\ll(H) = Lin_<(H') = Lin_\ll(H')$ and we say that H and H' are equivalent. The transformation of the original HMSC H proceeds in two steps.

Step 1 We modify each BMSC M in a state of H such that each process is covered with a coregion open on both sides, while BMSCs represented by the resulting HMSC remain the same as those represented by H: the same events on the same processes with the same visual and causal orders. We use general orderings and two fresh gate names pre, suc to induct the same

[2] In fact, the linearization corresponding to the mentioned execution has to be in $Lin_<(L(\sigma'))$.

visual (and hence also causal) orders. The definition of concatenation implies that, on a process p, all events of BMSCs preceding M in some run of H are visually ordered before all events of M (except those in a top-open coregion). This relation is preserved using the gate $p.\overline{pre}$. Similarly, all events of M (except those in a bottom-open coregion) are visually ordered before all events of BMSCs succeeding M in some run of H. This relation is preserved using the gate $p.\underline{suc}$.

More precisely, the general ordering $\prec_{C'}$ of a coregion C' open on both sides and covering a process p is defined as the least relation satisfying the following conditions (where $\mathbf{G}' = \mathbf{G} \cup \{pre, suc\}$ and $<$ refers to the original visual order of M):

- For every $e \in (E \cup p.\overline{\mathbf{G}}), f \in (E \cup p.\underline{\mathbf{G}})$, if $e < f$, then $(e,f) \in \prec_{C'}$.
- For every $e \in E$, if e is not in a top-open coregion in M, then $(p.\overline{pre}, e) \in \prec_{C'}$.
- For every $e \in E$, if e is not in a bottom-open coregion in M, then $(e, p.\underline{suc}) \in \prec_{C'}$.
- $p.\overline{suc} \times (E \cup p.\underline{\mathbf{G}'}) \subseteq \prec_{C'}$.
- $(E \cup p.\overline{\mathbf{G}'}) \times p.\underline{pre} \subseteq \prec_{C'}$.

Step 2 We restrict the general orderings to induce visual orders equivalent to the original causal orders. Due to Definition 3, it is sufficient to generally order pairs (e,f) such that $e < f$ and $f \in E_S$ (where $<$ refers to the original visual order). Formally, we replace every general ordering $\prec_{C'}$ computed in the previous step with $\prec_{C'} \cap ((\mathbf{P}.\overline{\mathbf{G}'} \cup E) \times (\mathbf{P}.\underline{\mathbf{G}'} \cup E_s))$.

5 Trace-race detection problem for HMSCs

This section studies decidability and complexity of the *trace-race detection problem*, i.e. the problem whether a given HMSC (with open coregions) contains a trace-race. We assume that each state of a given HMSC $H = (S, \to, s_0, S_F, L, \mathbf{L})$ appears on some run of H and it is labelled with a race-free BMSC. Recall that H contains a trace-race if and only if there is a run σ such that the BMSC $L(\sigma)$ contains a race. There is such a run if and only if there is a path $\pi = s_0 s_1 s_2 \ldots s_k$ where $L(s_0 \ldots s_{k-1})$ is a race-free BMSC containing an event e and $L(s_k)$ is a race-free BMSC containing an event f such that $L(\pi)$ contains a race between e and f. Due to Theorem 1, one can easily check whether a given path $\pi = s_0 s_1 s_2 \ldots s_k$ meets these conditions. However, it does not solve the trace-race detection problem as there could be infinitely many paths starting in s_0.

Our detection technique relies on a precise characterization of races appearing in concatenation $M_1 \cdot M_2$ of two race-free BMSCs. For this, we need two new functions returning sets of *joined gates*. Intuitively, a joined gate $p.g$ for a concatenation $M_1 \cdot M_2$ is a reference to a gate that appears as a bottom gate $p.\underline{g}$ in M_1 and is identified with the corresponding top gate $p.\overline{g}$ of M_2 during concatenation. We denote by $p.\mathbf{G}$ and $\mathbf{P}.\mathbf{G}$ the sets all joined gates over gate name space $\mathbf{G}$ and the process p or all processes $\mathbf{P}$, respectively.

Definition 11 Given a BMSC $M = (\mathbf{P}, E_S, E_R, P, \{<_p\}_{p \in \mathbf{P}}, \mathbf{M}, \mathbf{C}, \{\prec_C\}_{C \in \mathbf{C}})$ over gate name space $\mathbf{G}$, we define two functions $\downarrow(), \uparrow() : (E_S \cup E_R) \to 2^{\mathbf{P}.\mathbf{G}}$ as $\downarrow(e) = \{p.g \in \mathbf{P}.\mathbf{G} \mid e < p.\underline{g}\}$ and $\uparrow(e) = \{p.g \in \mathbf{P}.\mathbf{G} \mid p.\overline{g} < e\}$.

In the context of a concatenation $M_1 \cdot M_2$, $\downarrow(e)$ and $\uparrow(e)$ always refer to the values of these functions in the BMSC M_i (where $i \in \{1,2\}$) originally containing the event e. The characteri-

zation of races in $M_1 \cdot M_2$ is formulated in Lemma 3. The lemma assumes that the two race-free BMSCs M_1, M_2 are in the *special form* of Section 4, where all events and gates on each process are covered by a coregion open on both sides. Note that every BMSC can be converted to this form by the first step of the transformation presented in the previous section.

Lemma 3 *Let $M_1 \cdot M_2$ be a concatenation of two race-free BMSCs M_1, M_2 in the special form, e be an event of M_1, and f be an event of M_2 such that $P(e) = P(f) = p$. Then e and f are in race if and only if all the following conditions hold.*

1. f *is a receive event*
2. $\downarrow(e) \cap \uparrow(f) \cap p.\mathbf{G} \neq \emptyset$
3. $\downarrow(e) \cap \uparrow(\mathscr{M}^{-1}(f)) = \emptyset$
4. $label(e) = label(f) \implies \downarrow(\mathscr{M}^{-1}(e)) \cap \uparrow(\mathscr{M}^{-1}(f)) = \emptyset$
5. $\forall$ *receive events f' of M_2 such that $f' < \mathscr{M}^{-1}(f)$:* $label(e) = label(f') \implies \downarrow(\mathscr{M}^{-1}(e)) \cap \uparrow(\mathscr{M}^{-1}(f')) = \emptyset$

The precondition $P(e) = P(f)$ is not a serious restriction thanks to Lemma 2. The characterization says that to decide whether an event e of M_1 is in race with some event of M_2, one needs to know only $label(e)$, $\downarrow(e)$ and, if e is a receive action, then also $\downarrow(\mathscr{M}^{-1}(e))$. Triples $(label(e), \downarrow(e), \downarrow(\mathscr{M}^{-1}(e)))$ for receive events e and $(label(e), \downarrow(e), \emptyset)$ for send events e are called *footprints* of M_1. Note that the number of footprints for a fixed set of processes $\mathbf{P}$ and a gate name space $\mathbf{G}$ is bounded by $2 \cdot |\mathbf{P}|^2 \cdot 2^{|\mathbf{P}| \cdot |\mathbf{G}|} \cdot 2^{|\mathbf{P}| \cdot |\mathbf{G}|}$. Extending function P to labels as $P(p!q) = P(p?q) = p$, Lemma 3 can be reformulated as follows:

Lemma 4 *Let M_1 and M_2 be two race-free BMSCs in the special form. The concatenation $M_1 \cdot M_2$ contains a race if and only if there is a receive event f in M_2 and a footprint (l, F, F') of M_1 such that all the following conditions hold.*

1. $P(l) = P(f) = p$
2. $F \cap \uparrow(f) \cap p.\mathbf{G} \neq \emptyset$
3. $F \cap \uparrow(\mathscr{M}^{-1}(f)) = \emptyset$
4. $l = label(f) \implies F' \cap \uparrow(\mathscr{M}^{-1}(f)) = \emptyset$
5. $\forall$ *receive events f' of M_2 such that $f' < \mathscr{M}^{-1}(f)$:* $l = label(f') \implies F' \cap \uparrow(\mathscr{M}^{-1}(f')) = \emptyset$

Now we return to the observation from the beginning of this section. Let s, s' be two states of the HMSC H such that $(s, s') \in \rightarrow$. If we have the set of footprints of all BMSCs of the form $L(\pi)$ where π is a path leading from s_0 to s, we are able to decide whether any concatenation $L(\pi) \cdot L(s')$ contains a race. Moreover, we can effectively compute the set of footprints of all BMSCs of the form $L(\pi).L(s')$.

Hence, with each state s of the HMSC H we associate the set of all footprints of all BMSCs corresponding to paths starting in s_0 and leading to s. These sets of associated footprints can be easily computed using the fixpoint approach. If no race is detected during the computation, then the HMSC is trace-race-free. The precise algorithm, its complexity analysis, and complexity analysis of the trace-race detection problem can be found in [ŘSSH09].

Theorem 2 *Given an HMSC $H = (S, \rightarrow, s_0, S_f, L, \mathscr{L})$ (with open coregions and gates) over a gate name space $\mathbf{G}$, the problem whether H contains a trace-race is decidable in time $\mathscr{O}(|S|^2 \cdot b^3 \cdot |\mathbf{P}|^2 \cdot 2^{2 \cdot |\mathbf{P}| \cdot (|\mathbf{G}|+2)})$, where b is the size of the largest BMSC in $\mathbf{L}$. Hence, the problem is in* P *if the number of processes and gates is fixed.*

Theorem 3 *The trace-race detection problem is* PSPACE*-complete.*

6 Conclusions

We have introduced two new notions for HMSCs: an extension of the formalism with *open coregions* and a new race condition for HMSCs called *trace-race*. Definitions of race and trace-race directly imply that every trace-race-free HMSC is also race-free. We have shown that every race-free HMSC can be translated into an equivalent trace-race-free HMSC using open coregions, where by equivalence we mean that the two HMSCs represent sets of BMSCs with identical linearizations. Hence, trace-race-free HMSCs with open coregions are as expressive as race-free HMSCs with open coregions (and we conjecture that trace-race-free HMSCs with open coregions are in fact strictly more expressive than race-free HMSCs without open coregions). While the race detection problem is undecidable even for HMSCs without coregions [MP99], we have demonstrated that the trace-race detection problem is decidable (and PSPACE-complete) for HMSCs with open coregions. Therefore, HMSCs with open coregions and the trace-race notion appear as good candidates for tractable analysis of race ambiguities in scenario based designs.

The trace-race detection algorithm is implemented in *Sequence Chart Studio*, a Microsoft Visio add-on available at `http://scstudio.sourceforge.net/`. The studio currently supports HMSCs with closed coregions only (a support of open coregions is planned too).

Acknowledgements: We would like to thank Philippe Darondeau for an important hint.

Bibliography

[AHP96] R. Alur, G. Holzmann, D. Peled. An Analyzer for Message Sequence Charts. In *TACAS'96*. LNCS, pp. 35–48. Springer, 1996.

[DGH08] P. Darondeau, B. Genest, L. Hélouët. Products of Message Sequence Charts. In *FOSSACS 2008*. LNCS 4962, pp. 458–473. Springer, 2008.

[GH07] T. Gazagnaire, L. Hélouët. Event Correlation with Boxed Pomsets. In *FORTE 2007*. LNCS 4574, pp. 160–176. Springer, 2007.

[ITU04] ITU Telecommunication Standardization Sector - Study group 17. ITU Recommandation Z.120, Message Sequence Charts (MSC). 2004.

[MP99] A. Muscholl, D. Peled. Message Sequence Graphs and Decision Problems on Mazurkiewicz Traces. In *MFCS'99*. LNCS 1672, pp. 81–91. Springer, 1999.

[MP00] A. Muscholl, D. Peled. Analyzing Message Sequence Charts. In *SAM 2000: Proceedings of the 2nd Conference on MSC and SDL*. Pp. 3–17. 2000.

[Pra86] V. R. Pratt. Modeling concurrency with partial orders. *International Journal of Parallel Programming* 15(1):33–71, 1986.

[RGG96] E. Rudolph, P. Graubmann, J. Grabowski. Tutorial on Message Sequence Charts. *Computer Networks and ISDN Systems* 28(12):1629–1641, 1996.

[ŘSSH09] V. Řehák, P. Slovák, J. Strejček, L. Hélouët. Decidable Race Condition for HMSC. Technical report FIMU-RS-2009-10, Faculty of Informatics, Masaryk Univ., 2009.

Electronic Communications of the EASST
Volume of the Pre-proceedings of GT-VMT (2010)

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

A lightweight abstract machine for interaction nets

Abubakar Hassan, Ian Mackie and Shinya Sato

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

A lightweight abstract machine for interaction nets

Abubakar Hassan[1], **Ian Mackie**[2] **and Shinya Sato**[3]

[1] Department of Informatics, University of Sussex, Falmer, Brighton BN1 9QJ, UK

[2] LIX, CNRS UMR 7161, École Polytechnique, 91128 Palaiseau Cedex, France

[3] Himeji Dokkyo University, Faculty of Econoinformatics, 7-2-1 Kamiohno, Himeji-shi, Hyogo 670-8524, Japan

Abstract: We present a new abstract machine for interaction nets and demonstrate that an implementation based on the ideas is significantly more efficient than existing interaction net evaluators. The machine, which is founded on a chemical abstract machine formulation of interaction nets, is a simplification of a previous abstract machine for interaction nets. This machine, together with an implementation, is at the heart of current work on using interaction nets as a new foundation as an intermediate language for compiler technology.

Keywords: Interaction nets, programming languages, abstract machine

1 Introduction

Interaction nets [7] are a graphical model of computation. It is possible to program with interaction nets [6, 9] and they also serve as an intermediate language for implementing other programming languages. Some examples are encodings of λ-calculus, and simple functional programming languages (amongst others, see for instance [2, 4, 8]).

One reason why they have been very successful at implementing other programming languages is that a compilation must explain all the components of a computation. What is rare, is that the compilation can give something back, and this has been observed with the encodings on the λ-calculus where new strategies for reduction have been found. One of the reasons for this is because interaction nets naturally capture sharing, indeed one has to work hard to simulate reduction strategies where duplication of work takes place.

In [3] a calculus was given which provided a foundation for the operational understanding of interaction nets. This calculus led to the development of an abstract machine [10], which in turn led to a very efficient implementation of interaction nets.

Recently, there have been new developments in the foundations for a calculus of interaction nets. The purpose of this paper is to outline these ideas which led to the main contribution of the paper which is an abstract machine founded on the new calculus. This in turn has led to the development of new implementations of interaction nets which are the most efficient that we are aware of to date.

One of the main hopes of this work is that it provides a new foundation for a research programme to build implementations of programming languages through interaction nets: an improvement in the implementation technology for nets will have an impact on all the compilers

developed.

The main contributions of this paper are:

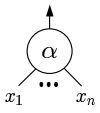
- We define a new term calculus of interaction nets. The novelty is that the notion of substitution is simplified in that it just replaces a name.
- We simplify and improve Pinto's abstract machine [10] by using this calculus. The main improvement is due to the fact that we no longer need to maintain lists of names, and consequently the transition rules become significantly more simple.
- We have built a prototype implementation based on the ideas. We demonstrate that we get a factor of ten improvement over previous implementations, and this implementation is thus the most efficient evaluator to date.

Overview. The rest of this paper is structured as follows. In the next section we review what we need about interaction nets. In Section 3 we give our new calculus. Section 4 gives the abstract machine, and gives studied properties of it. We conclude the paper in Section 5.

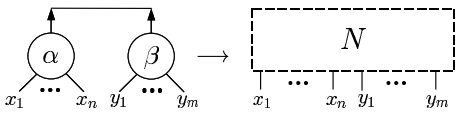
2 Interaction nets

Here we review the basic notions of interaction nets. We refer the reader to [7] for a more detailed presentation. Interaction nets are specified by the following data:

- A set Σ of *symbols*. Elements of Σ serve as *agent* (node) labels. Each symbol has an associated arity *ar* that determines the number of its *auxiliary ports*. If $ar(\alpha) = n$ for $\alpha \in \Sigma$, then α has $n+1$ *ports: n* auxiliary ports and a distinguished one called the *principal port*.



- A *net* built on Σ is an undirected graph with agents at the vertices. The edges of the net connect agents together at the ports such that there is only one edge at every port. A port which is not connected is called a *free port*. A set of free ports is called an *interface*.
- Two agents $(\alpha,\beta) \in \Sigma \times \Sigma$ connected via their principal ports form an *active pair* (analogous to a redex). An interaction rule $((\alpha,\beta) \longrightarrow N) \in \mathscr{R}_{in}$ replaces the pair (α,β) by the net N. All the free ports are preserved during reduction, and there is at most one rule for each pair of agents. The following diagram illustrates the idea, where N is any net built from Σ.



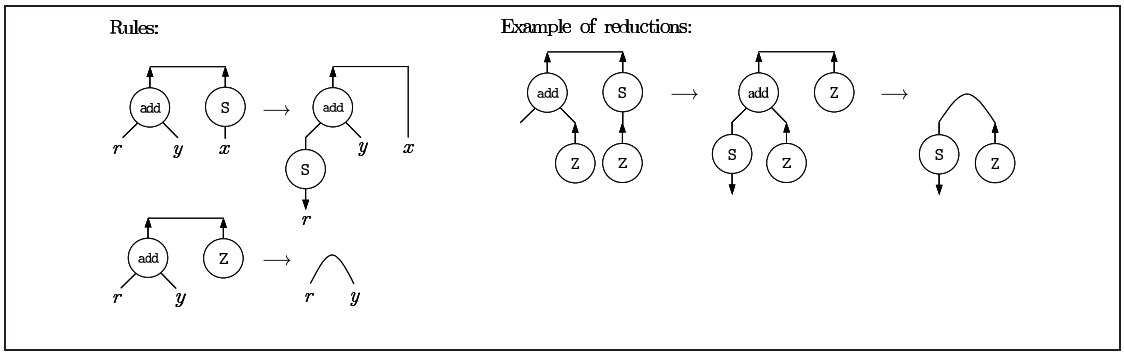


Figure 1: An example of a system of interaction nets

We use the relation $\longrightarrow$ for the one step reduction and $\longrightarrow^*$ for its transitive and reflexive closure. Interaction nets have the following property [7]:

Proposition 1 (Strong Confluence) *Let N be a net. If $N \longrightarrow N_1$ and $N \longrightarrow N_2$ with $N_1 \neq N_2$, then there is a net N_3 such that $N_1 \longrightarrow N_3$ and $N_2 \longrightarrow N_3$.*

Figure 1 shows a classical example of an interaction net system that encodes the addition operation. We can represent numbers using the agents S to represent the successor function $(n \mapsto n+1)$ and Z to represent the number 0. The left of the figure contains the two addition rules which we leave the reader to relate to the standard equational term rewriting system definition of addition. The right of the figure gives an example reduction sequence which shows how a net representing $0+1$ is reduced to 1 using the given rules.

2.1 The calculus for interaction nets

In this section we review the calculus for interaction nets proposed by Fernández and Mackie [3]. We begin by introducing a number of syntactic categories:

Names Let $\mathcal{N}$ be a set of *names* ranged over by $x, y, z, x_1, x_2, \ldots$. We write $\bar{x}, \bar{y}, \ldots$ for sequences of names. We assume $\mathcal{N}$ and Σ are disjoint.

Terms are built from Σ and $\mathcal{N}$ using the grammar: $t ::= x \mid \alpha(t_1, \ldots, t_n)$, where $t_1, \ldots, t_n$ are terms, $\alpha \in \Sigma$ and $ar(\alpha) = n$. If $ar(\alpha) = 0$, then we omit brackets and write just α. We use $t, s, u, \ldots$ to range over terms and $\bar{t}, \bar{s}, \bar{u}, \ldots$ over sequences of terms.

Equations have the form: $t = s$, where t and s are terms. Equations are elements of computation. Given $\bar{t} = t_1, \ldots, t_k$ and $\bar{s} = s_1, \ldots, s_k$, we write $\bar{t} = \bar{s}$ to denote the list $t_1 = s_1, \ldots, t_k = s_k$. We use $\Delta, \Theta, \ldots$ to range over multisets of equations.

Configurations have the form: $\langle\, \bar{t} \mid \Delta \,\rangle$, where $\bar{t}$ is a sequence of terms representing the interface of the net and Δ is a sequence of equations. All names occur at most twice in a configuration. We use $C_1, C_2, \ldots$ to range over configurations. Configurations that differ only on names are considered equivalent.

Interaction rules have the form: $\alpha(t_1,...,t_n) \bowtie \beta(s_1,...,s_k)$, where $\alpha(t_1,...,t_n)$ and $\beta(s_1,...,s_k)$ are terms. This notation for rules was introduced by Lafont [7] and we refer to it as Lafont's style. All names occur exactly twice in a rule, and there should be at most one rule between any pair of agents in $\mathscr{R}$. $\mathscr{R}$ is closed under symmetry, thus if $\alpha(\bar{t}) \bowtie \beta(\bar{s}) \in \mathscr{R}$ then $\beta(\bar{s}) \bowtie \alpha(\bar{t}) \in \mathscr{R}$.

Definition 1 (Bound names) If a name x occurs twice in a term t, then we say x is *bound*. We extend this notion to equations, sequences of terms, and multiset of equations.

The calculus consists of three reduction rules which reduce (valid) configurations.

Indirection:
$\langle\, \bar{t} \mid x = t, u = s, \Delta \,\rangle \longrightarrow_i \langle\, \bar{t} \mid u[t/x] = s, \Delta \,\rangle$ where x occurs in u,

Collect:
$\langle\, \bar{t} \mid x = t, \Delta \,\rangle \longrightarrow_c \langle\, \bar{t}[t/x] \mid \Delta \,\rangle$ where x occurs in $\bar{t}$,

Interaction:
$\langle\, \bar{t} \mid \alpha(\bar{t}_1) = \beta(\bar{t}_2), \Delta \,\rangle \longrightarrow_{\bowtie} \langle\, \bar{t} \mid \bar{t}_1 = \bar{s}^l, \bar{t}_2 = \bar{u}^l, \Delta \,\rangle$

where $\alpha(\bar{s}) \bowtie \beta(\bar{u}) \in \mathscr{R}$ and $\bar{s}^l$ and $\bar{u}^l$ are the result of replacing each occurrence of a bound name x for $\alpha(\bar{s}) \bowtie \beta(\bar{u})$ by a fresh name x^l respectively.

Example 1 *The example rules in Figure 1 can be represented using Lafont's style* [1] *as:*

$$\mathtt{add}(\mathtt{S}(x), y) \bowtie \mathtt{S}(\mathtt{add}(x, y)), \qquad \mathtt{add}(x, x) \bowtie \mathtt{Z}$$

The example net in Figure 1 can be represented using the configuration:

$$\langle\, a \mid \mathtt{add}(a, \mathtt{Z}) = \mathtt{S}(\mathtt{Z}) \,\rangle$$

and the following is a possible reduction sequence using the calculus rules above:

$$\begin{aligned}
\langle\, a \mid \mathtt{add}(a, \mathtt{Z}) = \mathtt{S}(\mathtt{Z}) \,\rangle \quad &\longrightarrow_{\bowtie} \quad \langle\, a \mid a = \mathtt{S}(x'), \mathtt{Z} = y', \mathtt{Z} = \mathtt{add}(x', y') \,\rangle \\
&\longrightarrow_c \quad \langle\, \mathtt{S}(x') \mid \mathtt{Z} = y', \mathtt{Z} = \mathtt{add}(x', y') \,\rangle \\
&\longrightarrow_i \quad \langle\, \mathtt{S}(x') \mid \mathtt{Z} = \mathtt{add}(x', \mathtt{Z}) \,\rangle \\
&\longrightarrow_{\bowtie} \quad \langle\, \mathtt{S}(x') \mid x' = x'', \mathtt{Z} = x'' \,\rangle \\
&\longrightarrow_c \quad \langle\, \mathtt{S}(x'') \mid \mathtt{Z} = x'' \,\rangle \\
&\longrightarrow_c \quad \langle\, \mathtt{S}(\mathtt{Z}) \mid \,\rangle
\end{aligned}$$

3 Refining the calculus

The calculus given in the previous section has nice properties and provides a simple static and dynamic semantics for interaction nets. However, the calculus introduces extra computational steps to reduce a given net to normal form. For example, the example net in Figure 1 reduces in two steps using the graphical setting while the same net reduces in six steps using the textual calculus (see Example 1). In this section, we answer the following question in the positive: can we optimise the calculus to obtain more efficient computations? The result of this question is our *lightweight* calculus which will form the basis of the *lightweight* abstract machine.

[1] see [7, 5] for a more detailed description of Lafont's style syntax

Interaction rules. The notation of Lafont's style generates (redundant) equations which will be reduced by the Indirection rule. In particular, if an auxiliary port of an interacting agent in a rule is connected to another auxiliary port, the application of an Interaction rule will generate an equation with a variable x on one side of the equation. Since all variables appear twice in a rule, x will eventually be eliminated using the Indirection rule. For example, this can be traced in Example 1 where the equation $\mathtt{z} = y'$ is generated in the configuration after applying the first rule $\mathtt{add}(\mathtt{S}(x),y) \bowtie \mathtt{S}(\mathtt{add}(x,y))$. In other words, the application of an Interaction rule to an active pair (α,β) where $\alpha(\bar{t_1},x,\bar{t_2}) \bowtie \beta(\bar{s_1}) \in \mathscr{R}$ will generate a configuration where an Indirection rule is applicable.

In order to eliminate the generation of redundant equations we introduce an alternative notation to represent interaction rules. We represent rules using the syntax: $lhs \longrightarrow rhs$ where lhs consists of an equation between the two interacting agents and rhs is a list of equations which represent the right-hand side net. All rules $\alpha(\bar{t}) \bowtie \beta(\bar{s})$ in Lafont's style can be written using our notation:

$$\alpha(\bar{t_1}) = \beta(\bar{s_1}) \longrightarrow \bar{t_1} = \bar{t}, \bar{s_1} = \bar{s} \qquad \text{where } \bar{t_1}, \bar{s_1} \text{ are meta-variables for terms.}$$

As a concrete example, the rule $\mathtt{add}(\mathtt{S}(x),y) \bowtie \mathtt{S}(\mathtt{add}(x,y))$ can be represented as

$$\mathtt{add}(t_1,t_2) = \mathtt{S}(u_1) \longrightarrow t_1 = \mathtt{S}(x), t_2 = y, u_1 = \mathtt{add}(x,y)$$

moreover we can simplify rules by replacing equals for equals. The above rule can be simplified to:

$$\mathtt{add}(t_1,t_2) = \mathtt{S}(u_1) \longrightarrow t_1 = \mathtt{S}(x), u_1 = \mathtt{add}(x,t_2)$$

Therefore we obtain a more efficient computation by using the notation of term rewriting systems.

Definition 2 (Lightweight interaction rules) A lightweight rule $r \in \mathscr{R}_{\mathrm{lt}}$ is of the form:

$$\alpha(t_1,...,t_n) = \beta(s_1,...,s_k) \longrightarrow \Delta$$

where $\alpha,\beta \in \Sigma$, $\mathrm{ar}(\alpha) = n, \mathrm{ar}(\beta) = k$, and $t_1,...,t_n,s_1,...,s_k$ are meta-variables for terms. Each meta-variable occurs exactly twice in a rule: once on the lhs and once on the rhs. The set $\mathscr{R}_{\mathrm{lt}}$ contains at most one rule between any pair of agents; $\mathscr{R}_{\mathrm{lt}}$ is closed under symmetry — if $\alpha(\bar{t}) = \beta(\bar{s}) \longrightarrow \Delta \in \mathscr{R}_{\mathrm{lt}}$ then $\beta(\bar{s}) = \alpha(\bar{t}) \longrightarrow \Delta \in \mathscr{R}_{\mathrm{lt}}$.

Indirection rules. Let us now examine the Indirection rule of the calculus which eliminates bound variables by means of variable substitution. The application of this rule will search through the list of terms to locate a term which contains an occurrence of a particular variable. In order to reduce the searching costs, Pinto's abstract machine [10], which is based on this calculus, attaches a list of variables to the head of every term. This again introduces management overheads, hence the increase in the number of operations required to perform rewirings.

Taking into consideration that every change of connection does not affect interactions directly, it turns out that we do not have to perform all substitutions eagerly. Therefore we decompose the Indirection rule into: *communication rules* that will replace just a name, and *substitution rule* that will perform other substitutions.

Definition 3 (Lightweight reduction rules) We define Lightweight reduction rules as follows:

Communication:
$\langle\, \bar{t} \mid x = t, x = u, \Delta \,\rangle \xrightarrow{com} \langle\, \bar{t} \mid t = u, \Delta \,\rangle$,

Substitution:
$\langle\, \bar{t} \mid x = t, u = s, \Delta \,\rangle \xrightarrow{sub} \langle\, \bar{t} \mid u[t/x] = s, \Delta \,\rangle$ where u is not a name and x occurs in u,

Collect:
$\langle\, \bar{t} \mid x = t, \Delta \,\rangle \xrightarrow{col} \langle\, \bar{t}[t/x] \mid \Delta \,\rangle$ where x occurs in $\bar{t}$,

Interaction:
$\langle\, \bar{t} \mid \alpha(\bar{t}_1) = \beta(\bar{t}_2), \Delta \,\rangle \xrightarrow{int} \langle\, \bar{t} \mid \Theta^l, \Delta \,\rangle$

where $\alpha(\bar{s}) = \beta(\bar{u}) \longrightarrow \Theta \in \mathscr{R}_{\mathrm{lt}}$ and Θ^l is the result of replacing each occurrence of a bound name x for Θ by a fresh name x^l and replacing each occurrence of $\bar{s}, \bar{u}$ by $\bar{t}_1, \bar{t}_2$ respectively.

We use just $\longrightarrow$ instead of $\xrightarrow{com}, \xrightarrow{sub}, \xrightarrow{col}, \xrightarrow{int}$ when there is no ambiguity. We define $C_1 \Downarrow C_2$ by $C_1 \longrightarrow^* C_2$ where C_2 is in normal form. From now on, we use $T, S, U, \ldots$ for non-variable terms.

Example 2 *Rules in Figure 1 can be represented as follows:*

$$\begin{array}{rcl} \mathtt{add}(x_1, x_2) = \mathtt{S}(y) & \longrightarrow & x_1 = \mathtt{S}(w), y = \mathtt{add}(w, x_2) \\ \mathtt{add}(x_1, x_2) = \mathtt{Z} & \longrightarrow & x_1 = x_2 \end{array}$$

and the following computation can be performed:

$$\begin{array}{rl} \langle\, a \mid \mathtt{add}(a, \mathtt{Z}) = \mathtt{S}(\mathtt{Z}) \,\rangle & \xrightarrow{int} \langle\, a \mid a = \mathtt{S}(w'), \mathtt{Z} = \mathtt{add}(w', \mathtt{Z}) \,\rangle \\ & \xrightarrow{col} \langle\, \mathtt{S}(w') \mid \mathtt{Z} = \mathtt{add}(w', \mathtt{Z}) \,\rangle \\ & \xrightarrow{int} \langle\, \mathtt{S}(w') \mid w' = \mathtt{Z} \,\rangle \\ & \xrightarrow{col} \langle\, \mathtt{S}(\mathtt{Z}) \mid \,\rangle \end{array}$$

3.1 Properties of lightweight reduction rules

In this section, we present some properties of the lightweight reduction rules. First, we show that we can postpone the application of Collect rules as in Abramsky's Computational interpretations of linear logic [1].

Lemma 1 *If* $C_1 \xrightarrow{col} \cdot \xrightarrow{com} C_2$ *then* $C_1 \xrightarrow{com} \cdot \xrightarrow{col} C_2$.

Proof. Let $C_1 = \langle\, \bar{t} \mid x = t, u = y, y = v, \Delta \,\rangle \xrightarrow{col} \langle\, \bar{t}[t/x] \mid u = y, y = v, \Delta \,\rangle \xrightarrow{com} \langle\, \bar{t}[t/x] \mid u = v, \Delta \,\rangle = C_2$. Then, $C_1 \xrightarrow{com} \langle\, \bar{t} \mid x = t, u = v, \Delta \,\rangle \xrightarrow{col} C_2$. □

Lemma 2 *If* $C_1 \xrightarrow{col} \cdot \xrightarrow{sub} C_2$ *then* $C_1 \xrightarrow{sub} \cdot \xrightarrow{col} C_2$. □

Lemma 3 *If* $C_1 \xrightarrow{col} \cdot \xrightarrow{int} C_2$ *then* $C_1 \xrightarrow{int} \cdot \xrightarrow{col} C_2$. □

By Lemma 1, 2, 3, the following holds.

Lemma 4 *If* $C_1 \Downarrow C_2$ *then there is a configuration* C *such that* $C_1 \longrightarrow^* C \xrightarrow{col}{}^* C_2$ *and* C_1 *is reduced to* C *without the application of any* Collect *rule*. □

Next, we examine whether or not we can postpone the application of Substitution rules. Note that applying the Substitution rule to an equation does not generate any other equations which require the application of an Interaction rule. Therefore the following properties hold.

Lemma 5 *If* $C_1 \xrightarrow{sub} \cdot \xrightarrow{com} C_2$ *then* $C_1 \xrightarrow{com} \cdot \xrightarrow{sub} C_2$. □

Lemma 6 *If* $C_1 \xrightarrow{sub} \cdot \xrightarrow{int} C_2$ *then* $C_1 \xrightarrow{int} \cdot \xrightarrow{sub} C_2$ *or* $C_1 \xrightarrow{int} \cdot \xrightarrow{com} C_2$. □

By Lemma 4, 5 and 6 the following theorem holds.

Theorem 1 *If* $C_1 \Downarrow C_2$ *then there is a configuration* C *such that* $C_1 \longrightarrow^* C \xrightarrow{sub}{}^* \cdot \xrightarrow{col}{}^* C_2$ *and* C_1 *is reduced to* C *by applying only* Communication *and* Interaction *rules*. □

This theorem shows that all Interaction rules can be performed without applying Substitution rules. We define $C_1 \Downarrow_{\mathrm{ic}} C_2$ by $C_1 \longrightarrow^* C_2$ where C_2 is a $\{\xrightarrow{int}, \xrightarrow{com}\}$−normal form.

4 Lightweight abstract machine

In this section we define the Lightweight abstract machine which is based on the lightweight rewriting rules.

Definition 4 (Machine configuration) A configuration of our abstract machine state is given by a 5-tuple $(\Gamma \mid \phi \mid \bar{t} \mid \Theta \mid \Delta)$ where

- Γ is an environment which maps a variable to a term. We use $[]$ as an empty map and the following notation:

$$\Gamma[x \mapsto t](z) = \begin{cases} t & (z \text{ is } x) \\ \Gamma(z) & (\text{otherwise}) \end{cases}$$

- ϕ is a *connection map*. When $\phi(x)$ is undefined, we use the following notation:

$$\phi[x \leftrightarrow \bot](z) = \begin{cases} \text{undefined} & (z = x) \\ \phi(z) & (\text{otherwise}) \end{cases}$$

- $\bar{t}$ is a sequence of terms
- Θ is a sequence of error codes that are not executable

Δ is a sequence of equations which we also regard as codes. We write "$-$" for an empty sequence of codes.

In Figure 2 we give the semantics of the machine as a set of transitional rules of the form: $(\Gamma \mid \phi \mid \bar{t} \mid \Theta \mid \Delta) \Longrightarrow (\Gamma' \mid \phi' \mid \bar{t} \mid \Theta' \mid \Delta')$. The functions **interaction**$(S = T)$ and **error**$(S = T)$ are defined as follows:

$$\textbf{interaction}(S=T) = \begin{cases} \Delta_1 & (\text{when } \langle \mid S=T \mid \rangle \xrightarrow{int} \langle \mid \Delta_1 \mid \rangle), \\ - & (\text{otherwise}) \end{cases}$$

$$\textbf{error}(S=T) = \begin{cases} - & (\text{when } \langle \mid S=T \mid \rangle \xrightarrow{int} \langle \mid \Delta_1 \mid \rangle), \\ S=T & (\text{otherwise}) \end{cases}$$

For readability purposes we present the transitions in a table format. For example, the entry:

		Before	After
II.0	Connections	$\phi\ [x \leftrightarrow \bot]$	$\phi\ [x \leftrightarrow \bot]$
	Env.	$\Gamma\ [x \mapsto \bot]$	$\Gamma\ [x \mapsto U]$
	Code	$x = U,\ \Delta$	Δ

corresponds to:

$$(\Gamma\ [x \mapsto \bot] \mid \phi\ [x \leftrightarrow \bot] \mid \bar{t} \mid - \mid x = U,\ \Delta) \Longrightarrow (\Gamma\ [x \mapsto U] \mid \phi\ [x \leftrightarrow \bot] \mid \bar{t} \mid - \mid \Delta)$$

4.1 Correctness

In order to show the correctness of our abstract machine, we first define a decompilation function from configurations to terms. Several lemmas follow before the correctness theorem.

Definition 5 (Decompilation) We define a translation $\lfloor . \rfloor_{env}$ from an environment Γ into a multiset of equations as follows:

$$\begin{aligned} \lfloor [] \rfloor_{env} &\stackrel{\text{def}}{=} empty, \\ \lfloor \Gamma[x \mapsto t] \rfloor_{env} &\stackrel{\text{def}}{=} x = t, \lfloor \Gamma \rfloor_{env}. \end{aligned}$$

The function $\lfloor . \rfloor_{con}$ translates a connection map ϕ into a multiset of equations as follows:

$$\begin{aligned} \lfloor [] \rfloor_{con} &\stackrel{\text{def}}{=} empty, \\ \lfloor \phi[x \leftrightarrow y] \rfloor_{con} &\stackrel{\text{def}}{=} x = y, \lfloor \phi \rfloor_{con}. \end{aligned}$$

We write just $\lfloor . \rfloor$ instead of $\lfloor . \rfloor_{env}, \lfloor . \rfloor_{con}$ when there is no ambiguity.

The machine will stop when there is no executable code. These cases arise not only when the code sequence is empty, but also when names are included in both the domains of Γ and ϕ. We define the latter case as inconsistent:

		Before	After
I	Error	Θ	$\mathbf{error}(U = T), \Theta$
	Code	$U = T,\ \Delta$	$\mathbf{interaction}(U = T),\ \Delta$
II.0	Connections	$\phi\ [x \leftrightarrow \bot]$	$\phi\ [x \leftrightarrow \bot]$
	Env.	$\Gamma\ [x \mapsto \bot]$	$\Gamma\ [x \mapsto U]$
	Code	$x = U,\ \Delta$	Δ
II.c	Connections	$\phi\ [x \leftrightarrow y]$	$\phi\ [x \leftrightarrow \bot][y \leftrightarrow \bot]$
	Env.	$\Gamma\ [x \mapsto \bot][y \mapsto \bot]$	$\Gamma\ [x \mapsto \bot][y \mapsto U]$
	Code	$x = U,\ \Delta$	Δ
II.e	Connections	$\phi\ [x \leftrightarrow \bot]$	$\phi\ [x \leftrightarrow \bot]$
	Env.	$\Gamma\ [x \mapsto T]$	$\Gamma\ [x \mapsto \bot]$
	Code	$x = U,\ \Delta$	$T = U,\ \Delta$
II.–	Code	$U = x, \Delta$	$x = U,\ \Delta$
III.0_0	Connections	$\phi\ [x \leftrightarrow \bot][y \leftrightarrow \bot]$	$\phi[x \leftrightarrow y]$
	Env.	$\Gamma\ [x \mapsto \bot][y \mapsto \bot]$	$\Gamma\ [x \mapsto \bot][y \mapsto \bot]$
	Code	$x = y, \Delta$	Δ
III.0_c	Connections	$\phi\ [x \leftrightarrow \bot][y \leftrightarrow w]$	$\phi\ [x \leftrightarrow w][y \leftrightarrow \bot]$
	Env.	$\Gamma\ [x \mapsto \bot][y \mapsto \bot]$	$\Gamma\ [x \mapsto \bot][y \mapsto \bot]$
	Code	$x = y,\ \Delta$	Δ
III.0_e	Connections	$\phi\ [x \leftrightarrow \bot][y \leftrightarrow \bot]$	$\phi\ [x \leftrightarrow \bot][y \leftrightarrow \bot]$
	Env.	$\Gamma\ [x \mapsto \bot][y \mapsto U]$	$\Gamma\ [x \mapsto U][y \mapsto \bot]$
	Code	$x = y,\ \Delta$	Δ
III.c_0	Connections	$\phi\ [x \leftrightarrow z][y \leftrightarrow \bot]$	$\phi\ [x \leftrightarrow \bot][y \leftrightarrow z]$
	Env.	$\Gamma\ [x \mapsto \bot][y \mapsto \bot]$	$\Gamma\ [x \mapsto \bot][y \mapsto \bot]$
	Code	$x = y,\ \Delta$	Δ
III.c_c	Connections	$\phi\ [x \leftrightarrow z][y \leftrightarrow w]$	$\phi\ [x \leftrightarrow \bot][y \leftrightarrow \bot][z \leftrightarrow w]$
	Env.	$\Gamma\ [x \mapsto \bot][y \mapsto \bot]$	$\Gamma\ [x \mapsto \bot][y \mapsto \bot]$
	Code	$x = y, \Delta$	Δ
III.c_e	Connections	$\phi\ [x \leftrightarrow z][y \leftrightarrow \bot]$	$\phi\ [x \leftrightarrow \bot][y \leftrightarrow \bot][z \leftrightarrow \bot]$
	Env.	$\Gamma\ [x \mapsto \bot][y \mapsto U]$	$\Gamma\ [x \mapsto \bot][y \mapsto \bot][z \mapsto U]$
	Code	$x = y, \Delta$	Δ
III.e_0	Connections	$\phi\ [x \leftrightarrow \bot][y \leftrightarrow \bot]$	$\phi\ [x \leftrightarrow \bot][y \leftrightarrow \bot]$
	Env.	$\Gamma\ [x \mapsto T][y \mapsto \bot]$	$\Gamma\ [x \mapsto \bot][y \mapsto T]$
	Code	$x = y,\ \Delta$	Δ
III.e_c	Connections	$\phi\ [x \leftrightarrow \bot][y \leftrightarrow w]$	$\phi\ [x \leftrightarrow \bot][y \leftrightarrow \bot][w \leftrightarrow \bot]$
	Env.	$\Gamma\ [x \mapsto T][y \mapsto \bot]$	$\Gamma\ [x \mapsto \bot][y \mapsto \bot][w \mapsto T]$
	Code	$x = y,\ \Delta$	Δ
III.e_e	Connections	$\phi\ [x \leftrightarrow \bot][y \leftrightarrow \bot]$	$\phi\ [x \leftrightarrow \bot][y \leftrightarrow \bot]$
	Env.	$\Gamma\ [x \mapsto T][y \mapsto U]$	$\Gamma\ [x \mapsto \bot][y \mapsto \bot]$
	Code	$x = y,\ \Delta$	$T = U,\ \Delta$

Figure 2: Transitions $(\Gamma \mid \phi \mid \bar{t} \mid \Theta \mid \Delta) \Longrightarrow (\Gamma' \mid \phi' \mid \bar{t} \mid \Theta' \mid \Delta')$

Definition 6 (Consistency of a machine state) A state $(\Gamma \mid \phi \mid \bar{t} \mid \Theta \mid \Delta)$ is consistent iff

- $\langle\, \bar{t} \mid \lfloor\Gamma\rfloor, \lfloor\phi\rfloor, \Theta, \Delta \,\rangle$ is a configuration, thus every name occurs at most twice,
- for every $x \in \mathcal{N}$, x is not included in both domains of Γ and ϕ.

The following lemma shows that consistency is preserved during transitions:

Lemma 7 *Let M_1 be a consistent state. If $M_1 \Longrightarrow M_2$, then M_2 is also consistent.* □

Let M_1 and M_2 be two abstract machine states. We define $M_1 \Downarrow M_2$ by $M_1 \Longrightarrow^* M_2$ where M_2 is a $\Longrightarrow$ −normal form.

Lemma 8 *Let M_1 be a consistent state, If $M_1 \Downarrow (\Gamma \mid \phi \mid \bar{t} \mid \Theta \mid \Delta)$, then Δ is empty.*

Proof. There exists a transition which can be applied to an equation $t = s$ whenever $(\Gamma \mid \phi \mid \bar{t} \mid \Theta \mid t = s, \Delta)$ is consistent. □

Lemma 9 *Let M_1 be a consistent state $(\Gamma_1 \mid \phi_1 \mid \bar{t} \mid \Theta_1 \mid \Delta_1)$. If $M_1 \Longrightarrow (\Gamma_2 \mid \phi_2 \mid \bar{t} \mid \Theta_2 \mid \Delta_2)$, then one of the following holds:*

- $\langle\, \bar{t} \mid \lfloor\Gamma_1\rfloor, \lfloor\phi_1\rfloor, \Theta_1, \Delta_1 \,\rangle = \langle\, \bar{t} \mid \lfloor\Gamma_2\rfloor, \lfloor\phi_2\rfloor, \Theta_2, \Delta_2 \,\rangle$,
- $\langle\, \bar{t} \mid \lfloor\Gamma_1\rfloor, \lfloor\phi_1\rfloor, \Theta_1, \Delta_1 \,\rangle \xrightarrow{int} \langle\, \bar{t} \mid \lfloor\Gamma_2\rfloor, \lfloor\phi_2\rfloor, \Theta_2, \Delta_2 \,\rangle$,
- $\langle\, \bar{t} \mid \lfloor\Gamma_1\rfloor, \lfloor\phi_1\rfloor, \Theta_1, \Delta_1 \,\rangle \xrightarrow{com} \langle\, \bar{t} \mid \lfloor\Gamma_2\rfloor, \lfloor\phi_2\rfloor, \Theta_2, \Delta_2 \,\rangle$,
- $\langle\, \bar{t} \mid \lfloor\Gamma_1\rfloor, \lfloor\phi_1\rfloor, \Theta_1, \Delta_1 \,\rangle \xrightarrow{com} \cdot \xrightarrow{com} \langle\, \bar{t} \mid \lfloor\Gamma_2\rfloor, \lfloor\phi_2\rfloor, \Theta_2, \Delta_2 \,\rangle$. □

Theorem 2 *Let $\langle\, \bar{t} \mid \Delta \,\rangle$ be a configuration. If $(\square \mid \square \mid \bar{t} \mid - \mid \Delta)$ terminates at $(\Gamma \mid \phi \mid \bar{t} \mid \Theta \mid \Delta')$, then Δ' is empty and $\langle\, \bar{t} \mid \Delta \,\rangle \Downarrow_{ic} \langle\, \bar{t} \mid \lfloor\Gamma\rfloor, \lfloor\phi\rfloor, \Theta \,\rangle$.*

Proof. By Lemma 8, Δ' is empty. Since $(\Gamma \mid \phi \mid \bar{t} \mid \Theta \mid -)$ is consistent by Lemma 7, $\lfloor\Gamma\rfloor$ and $\lfloor\phi\rfloor$ cannot contain equations that are reducible using the Communication rule. Therefore, by Lemma 9, $\langle\, \bar{t} \mid \Delta \,\rangle \Downarrow_{ic} \langle\, \bar{t} \mid \lfloor\Gamma\rfloor, \lfloor\phi\rfloor, \Theta \,\rangle$. □

Definition 7 We define the operation `update` as follows:

- $\texttt{update}(\Gamma \mid \phi[x \leftrightarrow y] \mid \bar{t} \mid \Theta \mid -) = \texttt{update}(\Gamma[x/y] \mid \phi \mid \bar{t}[x/y] \mid \Theta \mid -)$,
- $\texttt{update}(\Gamma[x \mapsto s] \mid \square \mid \bar{t} \mid \Theta \mid -) = \texttt{update}(\Gamma[s/x] \mid \square \mid \bar{t}[s/x] \mid \Theta \mid -)$,
- $\texttt{update}(\square \mid \square \mid \bar{t} \mid \Theta \mid -) = \bar{t}$.

Each execution of `update` corresponds to an application of either Substitution or Collect rules. Therefore, we can show the following property:

Theorem 3 (Correctness) *Let $\langle\, \bar{t} \mid \Delta \,\rangle$ be a configuration. If $(\square \mid \square \mid \bar{t} \mid - \mid \Delta) \Downarrow (\Gamma \mid \phi \mid \bar{t} \mid \Theta \mid \Delta')$, then Δ' is empty and there is a reduction path such that $\langle\, \bar{t} \mid \Delta \,\rangle \Downarrow \langle\, \bar{u} \mid \Theta' \,\rangle$ where $\texttt{update}(\Gamma \mid \phi \mid \bar{t} \mid \Theta \mid -) = \bar{u}$.*

	AMINE	Light	AMINE/Light
255II	14.07	0.09	156.33
264II	50.02	0.14	357.29
256II	119.93	0.23	521.43
A 3 6	4.14	0.18	23.00
A 3 7	40.15	0.71	57.04
A 3 8	612.19	1.70	360.11

Table 1: The execution times in seconds on Linux PC (2.6GHz, Pentium 4, 512MByte)

Example 3 *The computation of* $\langle\, r \mid \mathtt{Add}(r,\mathtt{Z}) = \mathtt{S}(\mathtt{Z}) \,\rangle$ *is given below:*

$$
\begin{aligned}
&(\,[]\mid[]\mid r \mid - \mid \mathtt{Add}(r,\mathtt{Z}) = \mathtt{S}(\mathtt{Z})\,)\\
&\quad\Longrightarrow (\,[]\mid[]\mid r \mid - \mid r = \mathtt{S}(x), \mathtt{Z} = \mathtt{Add}(x,\mathtt{Z})\,) && \text{(I)}\\
&\quad\Longrightarrow (\,[r \mapsto \mathtt{S}(x)]\mid[]\mid r \mid - \mid \mathtt{Z} = \mathtt{Add}(x,\mathtt{Z})\,) && \text{(II.0)}\\
&\quad\Longrightarrow (\,[r \mapsto \mathtt{S}(x)]\mid[]\mid r \mid - \mid x = \mathtt{Z}\,) && \text{(I)}\\
&\quad\Longrightarrow (\,[r \mapsto \mathtt{S}(x)][x \mapsto \mathtt{Z}]\mid[]\mid r \mid - \mid -\,) && \text{(II.0).}
\end{aligned}
$$

$$
\begin{aligned}
&\mathtt{update}(\,[r \mapsto \mathtt{S}(x)][x \mapsto \mathtt{Z}]\mid[]\mid r \mid - \mid -\,)\\
&\quad = \mathtt{update}(\,[r \mapsto \mathtt{S}(\mathtt{Z})]\mid[]\mid r \mid - \mid -\,) = \mathtt{S}(\mathtt{Z}).
\end{aligned}
$$

4.2 Benchmark results

We compare the lightweight version with Pinto's implementation (AMINE). Both are written in C language. Table 1 shows execution times in seconds of our implementation and AMINE. The final column gives the ratio between the two. The first three input programs are applications of church numerals where $n = \lambda f.\lambda x.f^n x$ and $\mathtt{I} = \lambda x.x$. The encodings of these terms into interaction nets are given in [8]. The next programs compute the Ackermann function. The following rules are the interaction net encoding of the Ackermann function:

$\mathtt{Pred}(\mathtt{Z}) \bowtie \mathtt{Z}$, $\quad \mathtt{Dup}(\mathtt{Z},\mathtt{Z}) \bowtie \mathtt{Z}$,
$\mathtt{Pred}(x) \bowtie \mathtt{S}(x)$, $\quad \mathtt{Dup}(\mathtt{S}(a),\mathtt{S}(b)) \bowtie \mathtt{S}(\mathtt{Dup}(a,b))$,
$\mathtt{A}(r,\mathtt{S}(r)) \bowtie \mathtt{Z}$, $\quad \mathtt{A1}(\mathtt{Pred}(\mathtt{A}(\mathtt{S}(\mathtt{Z}),r)),r) \bowtie \mathtt{Z}$,
$\mathtt{A}(\mathtt{A1}(\mathtt{S}(x),r),r) \bowtie \mathtt{S}(x)$, $\quad \mathtt{A1}(\mathtt{Dup}(\mathtt{Pred}(\mathtt{A}(r1,r)),\mathtt{A}(y,r1)),r) \bowtie \mathtt{S}(y)$,

and A 3 5 means computation of $\langle\, r \mid \mathtt{A}(\mathtt{S}(\mathtt{S}(\mathtt{S}(\mathtt{S}(\mathtt{S}(\mathtt{Z})))),r) = \mathtt{S}(\mathtt{S}(\mathtt{S}(\mathtt{Z}))) \,\rangle$.

The results that we have obtained are better than previous implementation results, and allow substantially larger classes of functions to be executed very efficiently. Depending on the architecture used, these results will vary slightly. We however invite the reader to try some of these examples by downloading our implementation: http://www.interaction-nets.org/.

5 Conclusion

The aim of this paper is to report on current work on the foundations of the implementations of interaction nets. Specifically, we have presented a new implementation that is the most efficient to date. In the work where interaction nets are considered as an intermediate language for compilation, this work gives a speedup by a factor of ten or more.

Implementation work for interaction nets is currently being investigated very actively, and although this step is a considerable one, we believe that there is still much more to do. Our implementations are still very much prototype in nature, and no program optimisations have been included here. Future work will be directed towards developing stable and efficient implementations for both sequential and parallel architectures.

Bibliography

[1] S. Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 111:3–57, 1993.

[2] A. Asperti and S. Guerrini. *The Optimal Implementation of Functional Programming Languages*, volume 45 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1998.

[3] M. Fernández and I. Mackie. A calculus for interaction nets. In G. Nadathur, editor, *Proceedings of the International Conference on Principles and Practice of Declarative Programming (PPDP'99)*, number 1702 in LNCS, pages 170–187. Springer-Verlag, 1999.

[4] G. Gonthier, M. Abadi, and J.-J. Lévy. The geometry of optimal lambda reduction. In *Proceedings of the 19th ACM Symposium on Principles of Programming Languages (POPL'92)*, pages 15–26. ACM Press, Jan. 1992.

[5] A. Hassan, I. Mackie, and S. Sato. Interaction nets: programming language design and implementation. *ECEASST*, 10, 2008.

[6] A. Hassan, I. Mackie, and S. Sato. Compilation of interaction nets. *Electron. Notes Theor. Comput. Sci.*, 253(4):73–90, 2009.

[7] Y. Lafont. Interaction nets. In *Seventeenth Annual Symposium on Principles of Programming Languages*, pages 95–108, San Francisco, California, 1990. ACM Press.

[8] I. Mackie. YALE: Yet another lambda evaluator based on interaction nets. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 117–128. ACM Press, September 1998.

[9] I. Mackie. Towards a programming language for interaction nets. *Electronic Notes in Theoretical Computer Science*, 127(5):133–151, May 2005.

[10] J. S. Pinto. Sequential and Concurrent Abstract Machines for Interaction Nets. In J. Tiuryn, editor, *Proceedings of Foundations of Software Science and Computation Structures (FOSSACS)*, number 1784 in Lecture Notes in Computer Science, pages 267–282. Springer-Verlag, 2000.

Electronic Communications of the EASST
Pre-proceedings

De-/Re-constructing Model Transformation Languages

Eugene Syriani and Hans Vangheluwe

12 pages

No *ed(s) defined!
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

De-/Re-constructing Model Transformation Languages

Eugene Syriani and Hans Vangheluwe

McGill University, School of Computer Science, Montréal, Canada
{esyria,hv}@cs.mcgill.ca

Abstract: The diversity of today's model transformation languages makes it hard to compare their expressiveness and provide a framework for interoperability. De-constructing and then re-constructing model transformation languages by means of a unique set of most primitive constructs facilitates both. We thus introduce **T-Core**, a collection of primitives for model transformation. Combining **T-Core** with a (programming or modelling) language enables the design of model transformation formalisms. We show how basic and more advanced features from existing model transformation languages can be re-constructed using **T-Core** primitives.

Keywords: Transformation primitives, multi-paradigm model transformation

1 Introduction

A plethora of different rule-based model transformation languages and supporting tools exist today. They cover all (or a subset of) the well-known essential features of model transformation [SV09c]: *atomicity*, *sequencing*, *branching*, *looping*, *non-determinism*, *recursion*, *parallelism*, *back-tracking*, *hierarchy*, and *time*. For such languages, the semantics (and hence implementation) of a transformation rule consists of the appropriate combination of building blocks implementing primitive operations such as matching, rewriting, and often a validation of consistent application of the rule. The abovementioned essential features of transformation languages are achieved by implicitly or explicitly specifying "rule scheduling". Languages such as **ATL** [JK06], **FUJABA** [FNTZ00], **GReAT** [AKK+06], **MoTif** [SV09b], and **VIATRA** [VB07] include constructs to specify the order in which rules are applied. This often takes the form of a control flow language. Without loss of generality, we consider transformation languages where models are encoded as typed, attributed graphs.

The diversity of transformation languages makes it hard, on the one hand, to compare their expressiveness and, on the other hand, to provide a framework for interoperability (*i.e.*, meaningfully combining transformation units specified in different transformation languages). One approach is to express model transformation at the level of primitive building blocks. De-constructing and then re-constructing model transformation languages by means of a small set of most primitive constructs offers a common basis to compare the expressiveness transformation languages. It may also help in the discovery of novel, possibly in domain-specific, model transformation constructs by combining the building blocks in new ways. Furthermore, it allows implementers to focus on maximizing the efficiency of the primitives in isolation, leading to more efficient transformations overall. Lastly, once re-constructed, different transformation languages can seamlessly interoperate as they are built on the same primitives. This use of common primitives in turn allows for global as well as inter-rule optimization.

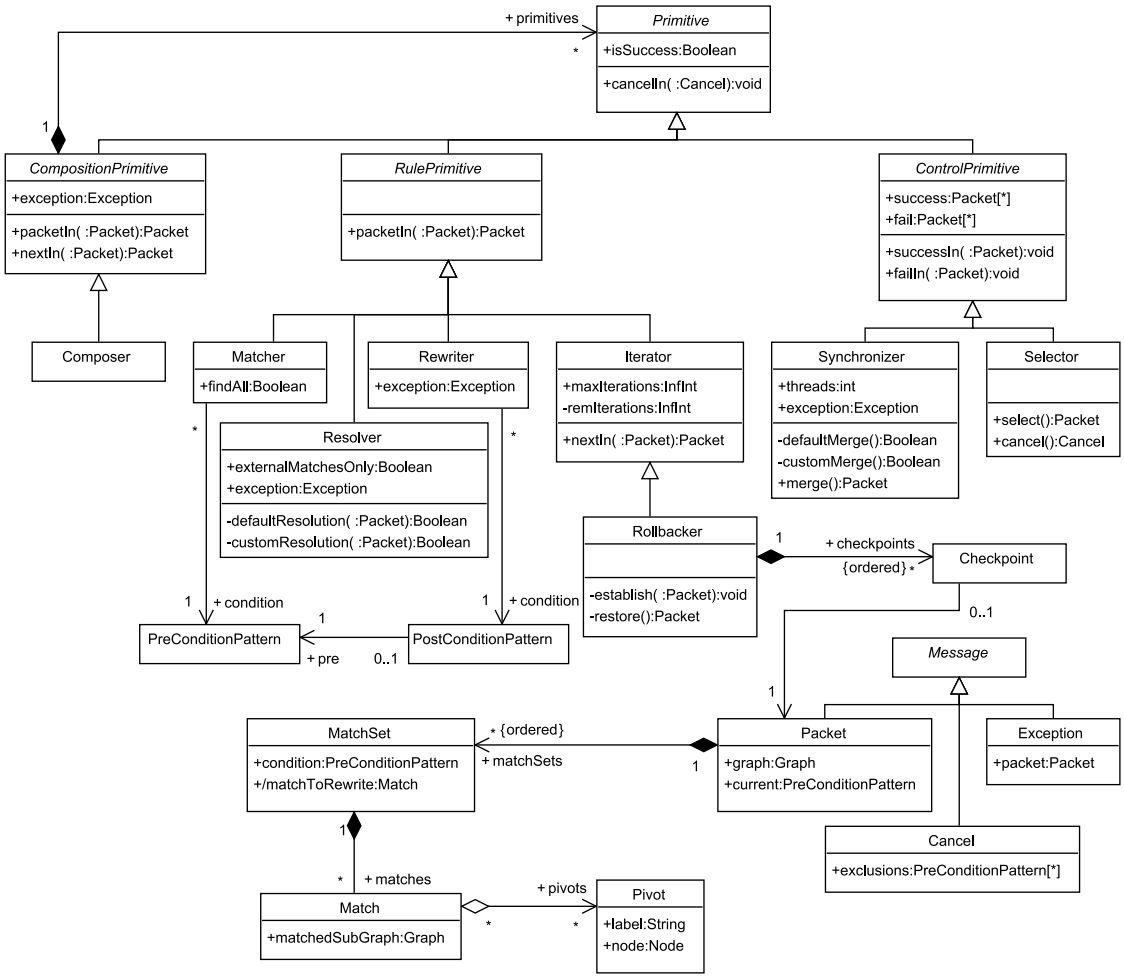


Figure 1: The **T-Core** module

We introduce **T-Core**, a collection of transformation language primitives for model transformation in Section 2. Section 3 motivates the choice of its primitives. Section 4 shows how transformation entities, common as well as more esoteric, can be re-constructed. Section 5 describes related work and Section 6 draws conclusions and presents directions for future work.

2 De-constructing Transformation Languages

We propose here a collection of model transformation primitives. The class diagram in Figure 1 presents the module **T-Core** encapsulating model transformation primitives. **T-Core** consists of eight primitive constructs (Primitive objects): a Matcher, Iterator, Rewriter, Resolver, Rollbacker, Composer, Selector, and Synchronizer. The first five are RulePrimitive elements and represent the building blocks of a single transformation unit. **T-Core** is not restricted to any form of specification of a transformation unit. In fact, we consider only PreConditionPatterns and PostCondi-

tionPatterns. For example, in rule-based model transformation, the transformation unit is a *rule*. The PreConditionPattern determines its applicability: it is usually described with a left-hand side (LHS) and optional negative application conditions (NACs). It also consists of a PostCondition-Pattern which imposes a pattern to be found after the rule was applied: it is usually described with a right-hand side (RHS). RulePrimitives are to be distinguished from the ControlPrimitives, which are used in the design of the rule scheduling part of the transformation language. A meaningful composition of all these different constructs in a Composer object allows modular encapsulation of and communication between Primitive objects.

Primitives exchange three different types of messages: Packet, Cancel, and Exception. A packet π represents the host model together with sufficient information for inter- and intra-rule processing of the matches. π thus holds the current model (graph in our case) graph, the match-Set, and a reference to the current PreConditionPattern identifying a MatchSet. A MatchSet refers to a condition pattern and contains the actual matches as well as a reference to the match-ToRewrite. Note that each MatchSet of a packet has a unique condition, used for identifying the set of matches. A Match consists of a sub-graph of the graph in π where each element is bound to an element in graph. Some elements (Nodes) of the match may be labelled as pivots, which allows certain elements of the model to be identified and passed between rules. A cancel message φ is meant to cancel the activity of an active primitive element (especially used in the presence of a Selector). Finally, specific exceptions χ can be explicitly raised, carrying along the currently processed packet π (π_ϕ is used to represent the empty packet).

All the primitive constructs can receive packets by invoking either their packetIn, nextIn, successIn, or failIn methods. The result of calling one of these methods sets the primitive in success or failure mode as recorded by the isSuccess attribute. Cancel messages are received from the cancelIn method. Next, we describe in detail the behaviour of the different methods supported by the primitive elements. A complete description can be found in [SV09a].

2.1 Matcher

Algorithm 1 Matcher.packetIn(π)

$M \leftarrow$ (all) matches of condition found in π.graph
if $\exists \langle$condition$, M' \rangle \in \pi$.matchSets **then**
  $M' \leftarrow M' \cup M$
else
  add $\langle$condition$, M \rangle$ to π.matchSets
end if
π.current $\leftarrow$ condition
isSuccess $\leftarrow M \neq \emptyset$
return π

The Matcher finds all possible matches of the condition pattern on the graph embedded in the packet it receives from its packetIn method. The transformation modeller may optimize the matching by setting the findAll attribute to false when he a priori knows that at most one match of this matcher will be processed in the overall transformation. The matching also considers the pivot mapping (if present) of the current match of π. After matching the graph, the Matcher stores the different matches in the packet as described in Algorithm 1. Some implementations may, for example, parametrize the Matcher by the condition pattern or embed it directly in the Matcher. The transformation units (*e.g.*, rules) may be compiled in pre/post-condition patterns or interpreted, but this is a tool implementation issue which we do not discuss here.

2.2 Rewriter

As described in Algorithm 2, the Rewriter applies the required transformation for its condition on the match specified in the packet it receives from its packetIn method. That match is consumed

by the Rewriter: no other operation can be further applied on it. Some validations are made in the Rewriter to verify, for example, that π.current.condition = condition.pre or that no error occurred during the transformation. In our approach, a modification (update or delete) of an element in $\{M | \langle \text{condition}.pre, M\rangle \in \pi.\text{matchSets}\}$ is automatically propagated to the other matches, if applicable.

Algorithm 2 Rewriter.packetIn(π)

```
if π is invalid then
    isSuccess ← false
    exception ← χ(π)
    return π
end if
M ← ⟨condition.pre, M⟩ ∈ π.matchSets
apply transformation on M.matchToRewrite
if transformation failed then
    isSuccess ← false
    exception ← χ(π)
    return π
end if
set all modified nodes in M to dirty
remove ⟨condition, M⟩ from π.matchSets
isSuccess ← true
return π
```

2.3 Iterator

The Iterator chooses a match among the set of matches of the current condition of the packet it receives from its packetIn method, as described in Algorithm 3. The match is chosen randomly in a Monte-Carlo sense, repeatable using sampling from a uniform distribution to provide a reproducible, fair sampling. When its nextIn method is called, the Iterator chooses another match as long as the maximum number of iterations maxIterations (possibly infinite) is not yet reached, as described in Algorithm 4. In the case of multiple occurrences of a MatchSet identified by π.current, the Iterator selects the last MatchSet.

Algorithm 3 Iterator.packetIn(π)

```
if ⟨π.current, M⟩ ∈ π.matchSets then
    choose m ∈ M
    M.matchToRewrite ← m
    remIterations ← maxIterations − 1
    isSuccess ← true
    return π
else
    isSuccess ← false
    return π
end if
```

Algorithm 4 Iterator.nextIn(π)

```
if ⟨π.current, M⟩ ∈ π.matchSets and remIterations > 0 then
    choose m ∈ M
    M.matchToRewrite ← m
    remIterations ← remIterations − 1
    isSuccess ← true
    return π
else
    isSuccess ← false
    return π
end if
```

2.4 Resolver

The Resolver resolves a potential conflict between matches and rewritings as described in Algorithm 5. For the moment, the Resolver detects conflicts in a simple conservative way: it prohibits any change to other matches in the packet (check for *dirty* nodes). However, it does not verify if a modified match is still valid with respect to its pre-condition pattern. The externalMatchesOnly attribute specifies whether the conflict detection should also consider matches from its match set identified by π.current or not. In the case of conflict, a default resolution function is provided but the user may also override it.

Algorithm 5 Resolver.packetIn(π)

```
for all condition c ∈ {c | ⟨c, M⟩ ∈ π.matchSets} do
    if externalMatchesOnly and c = π.current then
        continue
    end if
    for all match m ∈ M do
        if m has a dirty node then
            if customResolution(π) then
                isSuccess ← true
                return π
            else if defaultResolution(π) then
                isSuccess ← true
                return π
            else
                isSuccess ← false
                exception ← χ(π)
                return π
            end if
        end if
    end for
end for
isSuccess ← false
exception ← χ(π)
return π
```

2.5 Composer

The Composer serves as a modular encapsulation interface of the elements in its primitives list. When one

of its packetIn or nextIn methods is invoked, it is up to the user to manage subsequent method invocations to its primitives. Nevertheless, when the cancelIn method is called, the Composer invokes the cancelIn method of all its sub-primitives. This cancels the current action of the primitive object by resetting its state to its initial state. Cancelling happens only if it is actively processing a packet π such that the current condition of π is not in φ.exclusions, where φ is the received cancel message. In the case of a Matcher, since the current condition of the packet may not already be set, the cancelIn also verifies that the condition of the Matcher is not in the exclusions list. The interruption of activity can, for instance, be implemented as a pre-emptive asynchronous method call of cancelIn.

Additionally, more advanced primitive are included in **T-Core** but are not described here due to space constraints. A Rollbacker provides back-tracking capabilities to its transformation rule. A Selector is used when a choice needs to be made between multiple packets processed concurrently by different constructs. Also, a Synchronizer is used when multiple packets processed in parallel need to be synchronized.

3 T-Core: a minimal collection of transformation primitives

In the de-construction process of transformation languages into a collection of primitives, questions like "up to what level?" or "what to include and what to exclude?" arise. The proposed **T-Core** module answers these questions in the following way.

In a model transformation language, the smallest transformation unit is traditionally the *rule*. A rule is a complex structure with a declarative part and an operational part. The declarative part of a rule consists of the specification of the rule (*e.g.*, LHS/RHS and optionally NAC in graph transformation rules). However, **T-Core** is not restricted to any form of specification let it be rule-based, constraint-based, or function-based. In fact, some languages require units with only a pre-condition to satisfy, while other with a pre- and a post-condition. Some even allow arbitrary permutations of repetitions of the two. In **T-Core**, either a PreConditionPattern or both a Pre- and a PostConditionPattern must be specified. For example, a graph transformation rule can be represented in **T-Core** as a couple of a pre- and a post-condition pattern, where the latter has a reference to the former to satisfy the semantics of the interface K (in the $L \leftarrow K \rightarrow R$ algebraic graph transformation rules) and be able to perform the transformation. Transformation languages where rules are expressed bidirectionally or as functions are supported in **T-Core** as long as they can be represented as pre- and post-condition patterns.

The operational part of a rule describes how it executes. This operation is often encapsulated in the form of an algorithm (with possibly local optimizations). Nevertheless, it always consists of a *matching phase*, *i.e.*, finding instances of the model that satisfy the pre-condition and of a *transformation phase*, *i.e.*, applying the rule such that the resulting model satisfies the post-condition. **T-Core** distinguishes these two phases by offering a Matcher and a Rewriter as primitives. Consequently, the Matcher's condition only consists of a pre-condition pattern and the Rewriter then needs a post-condition pattern that can access the pre-condition pattern to perform the rewrite. Combinations of Matchers and Rewriters in sequence can then represent a sequence of simple graph transformation rules: *match-rewrite-match-rewrite*. Moreover, because of the separation of these two phases, more general and complex transformation units may be built,

such as: *match-match-match* or *match-match-rewrite-rewrite*. The former is a query where each Matcher filters the conditions of the query. The latter is a nesting of transformation rules. In this case, however, overlapping matches between different Matchers and then rewrites on the overlapping elements may lead to inconsistent transformations or even non-sense. This is why a Resolver can be used from **T-Core** to safely allow *match-rewrite* combinations.

The data structure exchanged between **T-Core** RulePrimitives in the form of packets contains sufficient information for each primitive to process it as described in the various algorithms in Section 2. The Match allows to refer to all model elements that satisfy a pre-condition pattern. The pivot mappings allow elements of certain matches to be bound to elements of previously matched elements. The pivot mapping is equivalent to passing parameters between rules as will be shown in the example in Section 4.1. The MatchSet allows to delay the rewriting phase instead of having to rewrite directly after matching.

Packets conceptually carry the complete model (optimized implementation may relax this) which allows concurrent execution of transformations. The Selector and the Synchronizer both permit to join branches or threads of concurrent transformations. Also, having separated the matching from the rewriting enables to manage the matches and the results of a rewrite by further operators. Advanced features such as iteration over multiple matches or back-tracking to a previous state in the transformation are also supported in **T-Core**.

Since **T-Core** is a low-level collection of model transformation primitives, combining its primitives to achieve relevant and useful transformations may involve a large number of these primitive operators. Therefore, it is necessary to provide a "grouping" mechanism. The Composer allows to modularly organize **T-Core** primitives. It serves as an interface to the primitives it encapsulates. This then enables scaling of transformations built on **T-Core** to large and complex model transformations designs.

T-Core is presented here as an open module which can be extended, through inheritance for example. One could add other primitive model transformation building blocks. For instance, a conformance check operator may be useful to verify if the resulting transformed model still conforms to its meta-model. It can be interleaved between sequences of rewrites or used at the end of the overall transformation as suggested in [KMW09]. We believe however that such new constructs should either be part of the (programming or modelling) language or the tool in which **T-Core** is integrated, to keep **T-Core** as primitive as possible.

4 Re-constructing Transformation Languages

Having de-constructed model transformation languages in a collection of model transformation primitives makes it easier to reason about transformation languages. In fact, properly combining **T-Core** primitives with an existing well-formed programming or modelling language allows us to re-construct some already existing transformation languages and even construct new ones [SV09a]. Figure 2 shows some examples of combinations of **T-Core** with other languages. Figure 2(a) and Figure 2(b) combine a subset of **T-Core** with a simple (programming) language which offers *sequencing*, *branching*, and *looping* mechanisms (as proposed in Böhm-Jocapini's *structured program theorem* [BJ66]). We will refer to such a language as an *SBL language*. The first combination only involves the Matcher and its PreConditionPattern, Packet messages to ex-

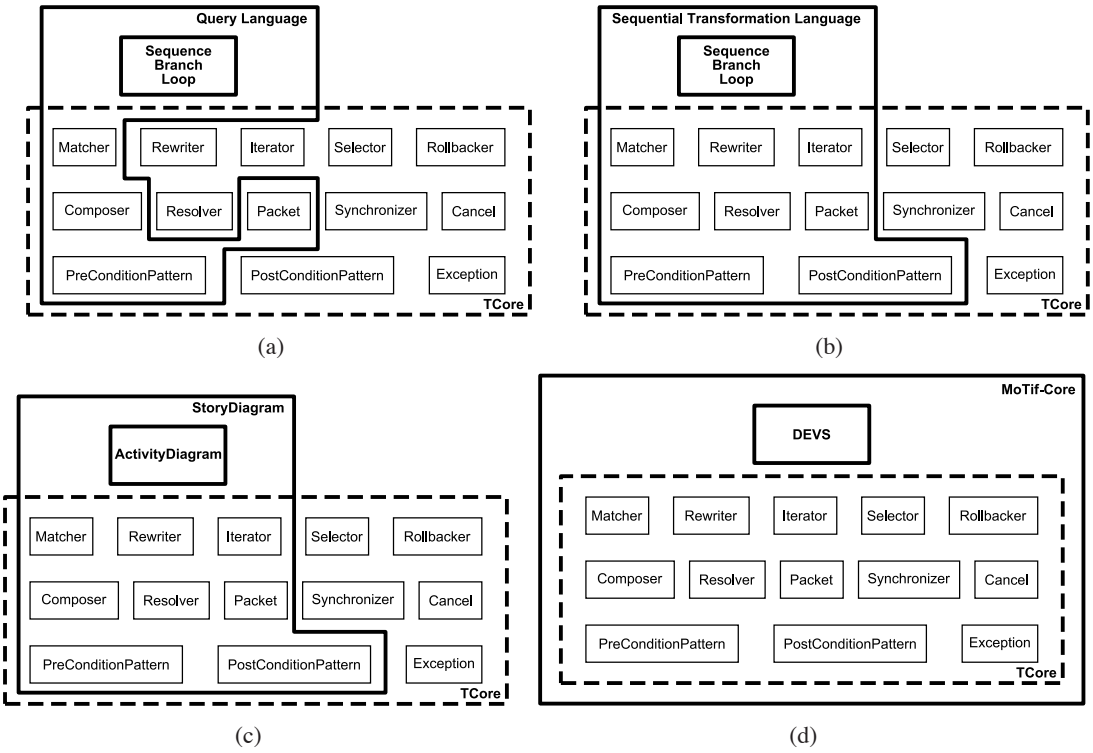


Figure 2: Combining **T-Core** with other languages allows to re-construct existing and new languages

change, and the Composer to organize the primitives. These **T-Core** primitives integrated in an SBL language lead to a *query language*. Since only matching operations can be performed on the model, they represent queries where the resulting packet holds the set of all elements (sub-graph) of the model (graph) that satisfy the desired pre-conditions. Including other **T-Core** primitives such as the Rewriter promotes the query language to a transformation language. Figure 2(b) enumerates the necessary **T-Core** primitives combined with an SBL language to design a complete sequential model transformation language. Replacing the SBL language by another one, such as UML Activity Diagrams in Figure 2(c), allows us to re-construct Story Diagrams [FNTZ00], for example, since they are defined as a combination of UML Activity and Collaboration Diagrams with graph transformation features. and the notion of timed model transformations when combined with a discrete-event modelling language [SV09a].

We now present the re-construction of two transformation features using the combination of an SBL language with **T-Core** as in Figure 2(b).

4.1 Re-constructing Story Diagrams

In the context of object-oriented reverse-engineering, the **FUJABA** tool allows the user to specify the content of a class method by means of Story Diagrams. A Story Diagram

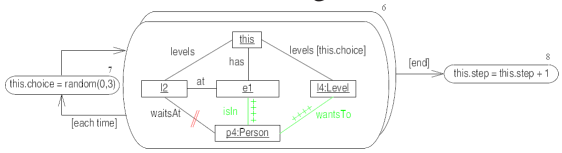


Figure 3: The **FUJABA** doSubDemo transformation showing a for-all Pattern and two statement activities

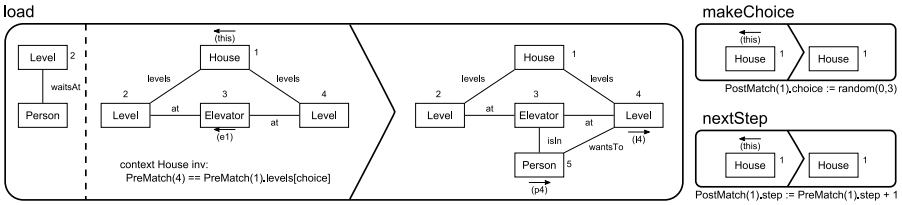


Figure 4: The three **MoTif** rules for the doSubDemo transformation

organizes the behaviour of a method with activities and transitions. An activity can be a Story Pattern or a statement activity. The former consists of a graph transformation rule and the latter is Java code. Figure 3 shows such a story diagram taken from the doDemo method example in [FNTZ00]. This snippet represents an elevator loading people on a given floor of a house who wish to go to another level. The rule in the pattern is specified in a UML Collaboration Diagram-like notation with objects and associations. Objects with implicit types (*e.g.*, this, l2, and e1) are *bound* objects from previous patterns or variables in the context of the current method. The Story Pattern 6 is a for-all Pattern. Its rule is applied on all matches found looping over the unbound objects (*e.g.*, p4, and l4). The outgoing transition labelled each time applies statement 7 after each iteration of the for-all Pattern. This activity allows the pattern to simulate random choices of levels for different people in the elevator. When all iterations have been completed, the flow proceeds with statement 8 reached by the transition labelled end. The latter activity simulates the elevator going one level up.

We now show how to re-construct this non-trivial story diagram transformation from an SLB language combined with **T-Core**. An instance of that combination is called a ***T-Core** model*. First, we design the rules needed for the conditions of rule primitives. Figure 4 describes the three necessary rules corresponding to the three Story Diagram activities. We use the syntax of **MoTif** [SV09b] where the central compartment is the LHS, the compartment on the right of the arrow head is the RHS and the compartment(s) on the left of dashed lines are the NAC(s). The concrete syntax for representing the pattern elements was chosen to be intuitively close enough to the **FUJABA** graphical representation. Numeric labels are used to uniquely identify different elements across compartments. Elements with an alpha-numeric label between parentheses denote pivots. A right-directed arrow on top of such a label depicts that the model element matched for this pattern element is assigned to a pivot (*e.g.*, p4 and l4). If the arrow is directed to the left, then the model element matched for this pattern element is bound to the specified pivot (*e.g.*, this and e1).

Algorithm 6 makeChoiceC.packetIn(π)

```
π ← makeChoiceM.packetIn(π)
if not makeChoiceM.isSuccess then
    isSuccess ← false
    return π
end if
π ← makeChoiceI.packetIn(π)
if not makeChoiceI.isSuccess then
    isSuccess ← true
    return π
end if
π ← makeChoiceW.packetIn(π)
if not makeChoiceW.isSuccess then
    isSuccess ← false
    return π
end if
π ← makeChoiceR.packetIn(π)
if not makeChoiceW.isSuccess then
    isSuccess ← false
    return π
end if
isSuccess ← true
return π
```

The **T-Core** model equivalent to the original doSubDemo transformation consists of a Composer doSubDemoC. It is composed of two Composers loadC and nextStepC each containing a

Matcher, an Iterator, a Rewriter, and a Resolver. The packetIn method of doSubDemoC first calls the corresponding method of loadC and then feeds the returned packet to the packetIn method of nextStepC. This ensures that the output packet of the overall transformation is the result of first loading all the Person objects and then moving the elevator by one step. makeChoiceC and nextStepC behave as simple transformation rules. Their packetIn method behaves as specified in Algorithm 6. First, the matcher is tried on the input packet. Note that the conditions of the matchers makeChoiceM and nextStepM are the LHSs of rules **makeChoice** and **nextStep**, respectively. If it fails, the composer goes into failure mode and the packet is returned. Then, the iterator chooses a match. Subsequently, the rewriter attempts to transform this match. Note that the conditions of the rewriters makeChoiceW and nextStepW are the RHSs of rules **makeChoice** and **nextStep**, respectively. If it fails, an exception is thrown and the transformation stops. Otherwise, the resolver verifies the application of this pattern with respect to other matches in the transformed packet. The behaviour of the resolution function will be elaborated on later. Finally, on a successful resolution, the resulting packet is output and the composer is put in success mode. loadC is the composer that emulates the for-all Pattern of the example. Algorithm 7 specifies that behaviour. After finding all matches with loadM (whose condition is the LHS and the NAC of rule **load**), the packet is forwarded to the iterator loadI to choose a match. The iteration is emulated by a loop with the failure mode of loadI as the breaking condition. Inside the loop, loadW rewrites the chosen match and loadR resolves possible conflicts. Then, the resulting packet is sent to makeChoiceC to fulfil the each time transition of the story digram. After that, the nextIn method of loadI is invoked with the new packet to choose a new match and proceed in the loop.

Algorithm 7 loadC.packetIn(π)

```
π ← loadM.packetIn(π)
if not loadM.isSuccess then
    isSuccess ← false
    return π
end if
π ← loadI.packetIn(π)
while true do
    if not loadI.isSuccess then
        isSuccess ← true
        return π
    end if
    π ← loadW.packetIn(π)
    if not loadW.isSuccess then
        isSuccess ← false
        return π
    end if
    π ← loadR.packetIn(π)
    if not loadR.isSuccess then
        isSuccess ← false
        return π
    end if
    π ← makeChoiceC.packetIn(π)
    π ← loadI.nextIn(π)
end while
isSuccess ← true
return π
```

Having seen the overall **T-Core** transformation model, let us inspect how the different Resolvers should behave in order to provide a correct and complete transformation. The first rewriter called is loadR and the first time it receives a packet is when a transformation is applied on one of the matches of loadM. Therefore each match consists of the same House (since it is a bound node), two Levels, an Elevator, and the associations between them. On the other hand, loadW only adds a Person and links it to a Level. Therefore the default resolution function of loadR applies successfully, since no matched element is modified nor is the NAC violated in any other match. The next resolver is makeChoiceR which is in the same loop as loadR. There, the House is conflicting with all the matches in the packet according to the conservative default resolution function. Note that makeChoiceM finds at most one match (the bound House element). However, makeChoiceW does not really conflict with matches found in loadM. We therefore specify a custom resolution function for makeChoiceR that always succeeds. The same applies for nextStepR.

4.2 Re-constructing amalgamated rules

In a recent paper, Rensink et al. claim that the *Repotting the Geraniums* example is inexpressible in most transformation formalisms [RK09]. The authors propose a transformation language that uses an amalgamation scheme for nested graph transformation rules. As we have seen in the previous example, nesting transformation rules is possible in **T-Core** and will be used to solve the problem. It consists of *repotting all flowering geraniums whose pots have cracked*. Figure 5 illustrates the two nested graph transformation rules involved and Algorithm 8 demonstrates the composition of primitive **T-Core** elements encoding these rules. baseM (with, as condition, the LHS of rule **base**) finds all broken pots containing a flowering geranium, given the input packet containing the input graph. The set of matches resulting in the packet are the combination of all flowering geraniums and their pot container. From then on starts the loop. First, baseI chooses a match. If one is chosen, baseW transforms this match and baseR resolves any conflicts. In this case, baseW only creates a new unbroken pot and assigns pivots. Therefore, baseR's resolution function always succeeds. In fact, the resolver is not needed here, but we include it for consistency. The innerC composer encodes the **inner** rule which finds the two bound pots and moves a flourishing flower in the broken pot to the unbroken one. In order to iterate over all the flowers in the broken pot, the innerC.packetIn method has the exact same behaviour as loadC.packetIn in Algorithm 7, with the exception of not calling a sub-composer (like makeChoiceC). Note that an always successful custom resolution function for innerR is required. After the Resolver successfully outputs the packet, the **inner** rule is applied. Then (and also if baseI had failed) baseM.packetIn is called again with the resulting packet. The loop ends when the baseM.packetIn method call inside the loop fails, which entails baseC to return the final packet in success mode.

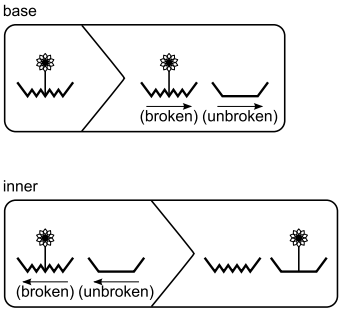


Figure 5: The transformation rules for the *Repotting Geraniums* example

Algorithm 8 baseC.packetIn(π)

```
π ← baseM.packetIn(π)
if not baseM.isSuccess then
    isSuccess ← false
    return π
end if
while true do
    π ← baseI.packetIn(π)
    if baseI.isSuccess then
        π ← baseW.packetIn(π)
        if not baseW.isSuccess then
            isSuccess ← false
            return π
        end if
        π ← baseR.packetIn(π)
        if not baseR.isSuccess then
            isSuccess ← false
            return π
        end if
        π ← innerC.packetIn(π)
    end if
    π ← baseM.packetIn(π)
    if not baseM.isSuccess then
        isSuccess ← false
        return π
    end if
end while
isSuccess ← true
return π
```

5 Related work

The closer work to our knowledge is [VJBB09]. In the context of global model management, the authors define a type system offering a set of primitives for model transformation. The advantage of our approach is that **T-Core** is a described here as a module and is thus directly implementable. We have recently incorporated **T-Core** with an asynchronous and timed mod-

elling language [SV09a] which allowed us to re-implement the two examples in Section 4 as well as others. Also, the approach described in [VJBB09], does not deal with exceptions at all. Nevertheless, their framework is able to achieve higher-order transformations, which we did not consider in this paper.

The **GP** graph transformation language [MP08] also offers transformation primitives. They however focus more on the scheduling of the rules then on the rules themselves. Their scheduling (control) language is an extension of an SBL language. Our approach is more general since much more complex scheduling languages (*e.g.*, allowing concurrent and timed transformation execution) can be integrated with **T-Core**. Although it performs very efficiently, the application area of **GP** is more limited, as it can not deal with arbitrary domain-specific models.

Other graph transformation tools, such as **VIATRA** [VB07] and **GReAT** [AKK+06], have their own virtual machine used as an API. In our approach, since the primitive operations are modelled, they are completely compatible with other existing model transformation frameworks.

6 Conclusion

In this paper, we have motivated the need for providing a collection of primitives for model transformation languages. We have defined **T-Core** which precisely describes each of these primitive constructs. The de-construction process of model transformation languages enabled us to reconstruct existing model transformation features by combining **T-Core** with, for example, an SBL language. This allowed us to compare different model transformation languages using a common basis.

Now that these primitives are well-defined, efficiently implementing each of them might lead to more efficient model transformation languages. Also, for future work, we would like to investigate how **T-Core** combined with appropriate modelling languages can express further transformation constructs. We would also like to investigate further on the notion of exceptions and error handling in the context of model transformation.

Bibliography

[AKK+06] A. Agrawal, G. Karsai, Z. Kalmar, S. Neema, F. Shi, A. Vizhanyo. The Design of a Language for Model Transformations. *SoSym* 5(3):261–288, September 2006.

[BJ66] C. Böhm, G. Jacopini. Flow diagrams, turing machines and languages with only two formation rules. *Communications of the ACM* 9(5):366–371, May 1966.

[FNTZ00] T. Fischer, J. Niere, L. Turunski, A. Zündorf. Story diagrams: A new graph rewrite language based on the Unified Modelling Language and Java. In Ehrig et al. (eds.), *Theory and Application of Graph Transformations*. LNCS 1764, pp. 296–309. Springer-Verlag, Paderborn (Germany), November 2000.

[JK06] F. Jouault, I. Kurtev. Transforming Models with ATL. In *MTiP'05*. LNCS 3844, pp. 128–138. Springer-Verlag, January 2006.

[KMW09] T. Kühne, E. V. H. Mezei, Gergely Syriani, M. Wimmer. Explicit Transformation Modelling. In *International workshop on Multi-Paradigm Modelling*. ECEASST. Denver (U.S.A.), October 2009.

[MP08] G. Manning, D. Plump. The GP Programming System. In *GT-VMT'08*. ECEASST, pp. 235–247. Budapest (Hungary), March 2008.

[RK09] A. Rensink, J.-H. Kuperus. Repotting the Geraniums: On Nested Graph Transformation Rules. In Margaria et al. (eds.), *GT-VMT'09*. EASST. York (UK), March 2009.

[SV09a] E. Syriani, H. Vangheluwe. De-/Re-constructing Model Transformation Languages. Technical report SOCS-TR-2009.8, McGill University, School of Computer Science, August 2009.

[SV09b] E. Syriani, H. Vangheluwe. *Discrete-Event Modeling and Simulation: Theory and Applications*. Chapter DEVS as a Semantic Domain for Programmed Graph Transformation. CRC Press, Boca Raton (USA), 2009.

[SV09c] E. Syriani, H. Vangheluwe. Matters of model transformation. Technical report SOCS-TR-2009.2, McGill University, School of Computer Science, March 2009.

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

[VJBB09] A. Vignaga, F. Jouault, M. C. Bastarrica, H. Brunelière. Typing in Model Management. In Paige (ed.), *Theory and Practice of Model Transformations (ICMT'09)*. LNCS 5563, pp. 197–212. Springer-Verlag, Zürich (Switzerland), June 2009.

Electronic Communications of the EASST
Volume of the Pre-proceedings of GT-VMT (2010)

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

Verification of Model Transformations

Bernhard Schätz

13 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

Verification of Model Transformations

Bernhard Schätz

fortiss GmbH
Guerickestr. 25, 80805 Mnchen, Germany
schaetz@fortiss.org

Abstract: With the increasing use of automatic transformations of models, the correctness of these transformations becomes an increasingly important issue. Especially for model transformation generally defined using abstract description techniques like graph transformations or declarative relational specifications, however, establishing the soundness of those transformations by test-based approaches is not straight-forward. We show how formal verification of soundness conditions over such declarative relational style transformations can be performed using an interactive theorem prover. The relational style allows a direct translation of transformations as well as associated soundness conditions into corresponding axioms and theorems. Using the Isabelle theorem prover, the approach is demonstrated for a refactoring transformation and a connectedness soundness condition.

Keywords: Model transformation, rule-based, verification, theorem prover

1 Motivation

The construction of increasingly sophisticated software products has led to widening gap between the required and supplied productivity in software development. To overcome the complexity of realistic software systems and thus increase productivity, current approaches increasingly focus on a *model-based* development using appropriate description techniques. Besides increasing efficiency, these transformations can offer consistency ensuring modification of models, ranging from refactoring steps to improve the architecture of a system to the consistent integration of standard behavior. However, with the increased use of transformation, the question of the correctness of transformations arises: How can we verify that the models constructed via transformation are 'well-formed' given a 'well-formed' source model, e.g., by ensuring that no relevant elements of the source model are absent in the target model. Obviously, testing is one possible way of ensuring the correctness of transformations. However, concepts like coverage etc. are not immediately transferable to model-transformations, especially if those are rules-based or declarative.

In the following, a approach for the verification of transformations is introduced, supporting the formal proof of properties over these transformations. The approach uses a declarative relational style to provide a transformation mechanism, implemented on the Eclipse/EMF Ecore platform, using a Prolog rule-based interpretation.

1.1 Related Approaches

Verification of model transformations has been specifically investigated for graph-based transformation techniques (e.g., [GGL+06] and [Str08]). In that respect, the presented approach is similar: The introduced transformation framework is used to describe graph transformations, using a relational calculus focused on basic constructs to manipulate nodes (elements) and edges (relations) of a conceptual model. A theorem prover based on on high-order logics is used to prove characteristics of the transformation by deducing properties of the target model from some properties of the source model.

In contrast to other graph-based approaches like MOFLON/TGG [KKS07], Viatra [VP04], or FuJaBa [GGL05], however, here the specification of transformations is not based on triple graph-grammars or graphical, rule-based descriptions, but uses a textual description based on a relational, declarative calculus. Therefore, in contrast to those approaches, the approach introduced here uses only a single formalism to describe basic transformations as well as their compositions.Furthermore, only a single homogenous formalism with two simple construction/deconstruction operators to describe the basic transformation rules and their composition; complex analysis or transformation steps can be easily modularized since there are no side-effects or incremental changes during the transformation. Thus, a specification can be immediately used for verification without complex translations; furthermore, proofs on the formal level more directly reflect intuitive reasoning about the transformation.

This homogeneity is especially important for verification since is drastically simplifies the construction of proofs: [GGL+06] focusses on TGG-based translation and therefore has to add substantial proof parts to model (and verify) the effective construction of correspondence graphs to describe the application of individual graph rules. Furthermore – due to that correspondence graph approach – there structural induction over the pre/post-models is used which is less convenient when if non-translation transformations are verified. Here, in contrast, induction over the transformation itself rather than the pre/post models is performed, thus having a more direct proof principle and avoiding the proof overhead of correspondence graphs and applicability conditions. Similarly, [Str08] also requires substantial effort to specify and verify correspondence graph and application conditions for single transformation rules as well their combination using *while* and *case* constructs. Thus, the application and ordering of rules provided *implicitly* by a TGG approach has to be verified *explicitly* and using rather different proof principles. In contrast, here, a more direct and homogenous form of proof is supported by the declarative rule-based style.

Another advantage of the presented approach is its capability to interpret loose characterizations of the resulting model, supporting the exploration of a set of possible solutions. By making use of the back-tracking mechanism provided by Prolog, alternative transformation results can not only be applied to automatically search for an optimized solution, e.g., balanced component hierarchies, using guiding metrics; the set of possible solutions can also be incrementally generated to allow the user to interactively identify and select the appropriate solution.

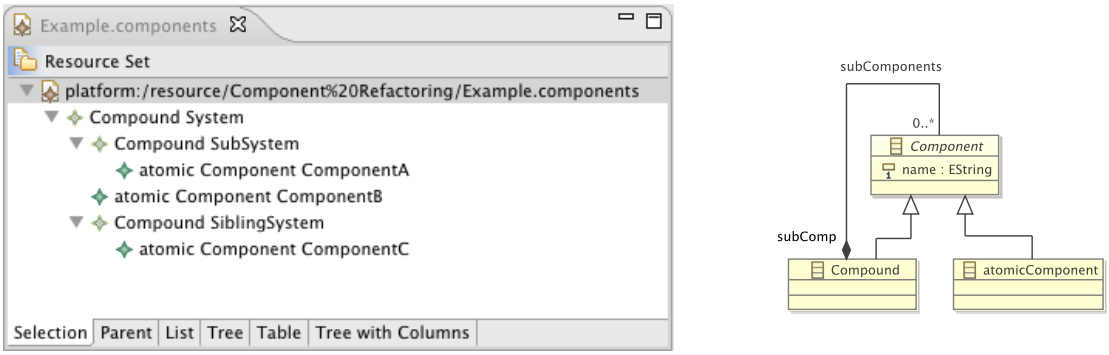


Figure 1: Example of Hierarchical Component Model and Corresponding Conceptual Model

1.2 Overview and Contribution

As the main contribution, an approach to *formally verify model transformations* is presented in the following sections. The approach is based on a transformation of EMF Ecore models using a completely *declarative relational* style in a *rule-based* fashion, introduced in [Sch08]. To provide such a form of transformations, the approach uses a term-based formalization of an EMF model as shown in Section 2. With this form of model representation, as shown in Section 3 transformations can be described as declarative relations in Prolog style, supporting rules similar to graph grammars as a specific description style.

Based on these previously established results, as *new contribution* in Section 4 the suitability of this declarative relational style of defining models and transformation rules for the verification of transformations is shown: The formalization of (meta-)models and transformation rules can be directly translated in representations suited for theorem provers for predicate logic like *Isabelle*; furthermore, due to the relational style correctness proofs of transformations can be performed by reasoning on the level of their specifications. Section 5 highlights some benefits and open issues.

2 Model Structure

To provide verified transformations of descriptions of systems, first the means of specifying a system in form of a system model is needed. The left-hand side of Figure 1 shows such a model, describing the hierarchical structure of the components of a system: the system System, consisting of subcomponents SubSystem, ComponentB, and SiblingSystem, the first and the last with subcomponents ComponentA and ComponentC, resp.[1]

To construct formalized descriptions of a system under development, a 'syntactic vocabulary' is needed. This conceptual model[2] characterizes all possible system models built from the *mod-*

[1] For simplification, here only components and their containment-relation is modeled; other typical aspects like interfaces or communication links are ignored.

[2] In the context of technologies like the Meta Object Facility, the class diagram-like definition of a conceptual model is generally called *meta model*.

eling concepts and their relations used to construct a description of a system; typically, class diagrams are used to describe them. The right-hand side of Figure 1 shows the corresponding conceptual model – with the concept of a Component with an attribute name and a subComp-relation – used to describe the architectural structure of a system.

2.1 Structure of the Model

The transformation framework provides mechanisms for a pure (i.e., side-effect free) declarative, rule-based approach to model transformation. To that end, the framework provides access to EMF Ecore-based models [SBPM07]. Based on the conceptual model, a system model consists of sets of elements (each described as a conceptual entity and its attribute values) and relations (each described as a pair of conceptual entities). To syntactically represent such a model, a Prolog term is used. Since these elements and relations are instances of classes and associations taken from an EMF Ecore model, the structure of the Prolog term – representing an instance of that model – is inferred from the structure of that model. The structure of the model is built using only simple elementary Prolog constructs, namely compound functor terms and list terms. To access a model, the framework provides construction predicates to deconstruct and reconstruct a term representing a model. [Sch08] describes the model in more detail.

A *model term* describes an instance of a EMF Ecore model. Each model term is a list of package terms, one for each packages of the EMF Ecore model. Each *package term*, in turn, describes the content of the package instance. It consists of a functor, identifying the package, with a sub-packages term, a classes terms, and an associations term as its argument. The sub-packages term describes the sub-packages of the package; it is a list of package terms.

The *classes term* describes the EClasses of the corresponding package. It is a list of class terms, one for each EClass. Each *class term* consists of a functor, identifying the class, and an elements term. An *elements term* describes the collection of objects instantiating this class, and thus is a list of element terms. Finally, an *element term* consists of a functor, identifying the class this object belongs to, with an entity identifying the element and attributes as arguments. Each of the attributes are atomic representations of the corresponding values of the attributes of the represented object. The entity is a regular atom, unique for each element term.

Similarly to an elements term, each *associations term* describes the associations, i.e., the instances of the EReferences of the EClasses, for the corresponding package. Again, it is a list of association terms, with each *association term* consisting of a functor, identifying the association, and an relations term, describing the content of the association. The *relations term* is a list of relation terms, each *relation term* consisting of a functor, identifying the relation, and the entity identificators of the related objects. In detail, the Prolog model term has the structure shown in Table 1 in the BNF notation with corresponding *non-terminals* and *terminals*.[3]

The functors of the compound terms are deduced from the EMF Ecore model: The functor of a PackageTerm from the name of the EPackage; the functor of a ClassTerm from the name of the EClass; the functor of an AssociationTerm from the name of the EReference. Similarly, the atoms of the attributes are deduced from the instance of the EMF Ecore model, which the model

[3] While actually a *ModelTerm* consists of a set of *PackageTerms*, here for simplification purposes only one *PackageTerm* is assumed.

ModelTerm	::=	*PackageTerm*
PackageTerm	::=	*Functor* (*PackagesTerm*, *ClassesTerm*, *AssociationsTerm*)
PackagesTerm	::=	[] | [*PackageTerm* (, *PackageTerm*)*]
ClassesTerm	::=	[] | [*ClassTerm* (, *ClassTerm*)*]
ClassTerm	::=	*Functor* (*ElementsTerm*)
ElementsTerm	::=	[] | [*ElementTerm* (, *ElementTerm*)*]
ElementTerm	::=	*Functor* (*Entity*(, *AttributeValue*)*)
Entity	::=	*Atom*
AttributeValue	::=	*Atom*
AssociationsTerm	::=	[] | [*AssociationTerm*(, *AssociationTerm*)*]
AssociationTerm	::=	*Functor* (*RelationsTerm*)
RelationsTerm	::=	[] | [*RelationTerm*(, *RelationTerm*)*]
RelationTerm	::=	*Functor* (*Entity*, *Entity*)

Table 1: The Prolog Structure of a Model Term

term is representing: The entity atom corresponds to the object identificator of an instance of a EClass, while the attribute corresponds to the attribute value of an instance of an EClass.

2.2 Construction Predicates

In a strictly declarative rule-based approach, the transformation is described in terms of a predicate, relating the models before and after the transformation. Therefore, mechanisms are needed in form of predicates to deconstruct a model into its parts as well as to construct a model from its parts. As the structure of the model is defined using only compound functor terms and list terms, only two forms of predicates are needed: union and composition operations.

2.2.1 List Construction

The(de)construction of lists is managed by means of the union predicate `union/3` with template[4] `union(?Left,?Right,?All)` such that `union(Left,Right,All)` is true if all elements of list `All` are either elements of `Left` or `Right`, and vice versa. Thus, e.g., `union([1, 3,5],R,[1,2,3,4,5])` succeeds with `R = [2,4]`.

2.2.2 Compound Construction

Since the compound structures used to build the model instances depend on the actual structure of the EMF Ecore model, only the general schemata used are described. Depending on whether a package, class/element, or association/relation is described, different schemata are used. In all three schemata the name of the package, class, or relation is used as the name of the predicate for the compound construction.

Packages For (de)construction of packages, package predicates of the form `package/4` are used with template `package(?Package,?Subpackages, ?Classes,?Associations)` where package is the name of the package (de)constructed. Thus,

[4] According to standard convention, arbitrary/input/output arguments of predicates are indicated by ?/+/−.

e.g., a package named Architecture in the EMF Ecore model is represented by the compound constructor `Architecture`. The predicate is true if `Package` consists of subpackages `Subpackages`, classes `Classes`, and associations `Associations`.

Classes and Elements For (de)construction of – non-abstract – classes/elements, class/element predicates of the form `class/2` and `class/N+2` are used where N is the number of the attributes of the corresponding class, with templates `class(?Class, ?Elements)` and `class(?Element,?Entity,?Attr1,...,?AttrN)` where class is the name of the class and element (de)constructed. Thus, e.g., the class named Compound in the EMF Ecore model in Figure 1 is represented by the compound constructor `Compound`. The class predicate is true if `Class` is the list of `Objects`; it is used to deconstruct a class into its list of objects, and vice versa. Similarly, the element predicate is true if `Element` is an `Entity` with attributes `Attr1`,...,`AttrN`; it can be used to deconstruct an element into its entity and attributes, to construct an element from an entity and attributes (e.g. to change the attributes of an element), or to construct a new element including its entity from the attributes. Thus, e.g., `Compound(Compounds,[Sys,Sub,Sib])` is used to construct a class `Compounds` from a list of objects `Sys`, `Sub`, and `Sib`. Similarly, `Compound(Sub,Subsys,"SubSystem")` is used to construct a new element `Sub` with entity `Subsys`, and name `"SubSystem"`.

Association and Relation Compounds For (de)construction of associations and relations, association and relation predicate of the form `association/2` and `association/3` are used with templates `association(?Association,?Relations)` and `association(?Relation,?Entity1,?Entity2)` where association is the name of the association and relation constructed/deconstructed. Thus, e.g., a relation named subComp in the EMF Ecore model in Figure 1 is represented by the compound constructor `subComp`. The relation predicate is true if `Association` is the list of `Relations`; it is generally used to deconstruct an association into its list of relations, and vice versa. Similarly, the relation predicate is true if `Relation` associates `Entity1` and `Entity2`; it is used to deconstruct a relation into its associated entities and vice versa. E.g., `subComp(subComps,[SubSys,SibSys])` is used to construct the subcomponent association `subComps` from the list of relations `SubSys` and `SibSys`. Similarly, `subComp(SubSys,Sub,Sys)` is used to construct relation `SubSys` with `Sub` being the subcomponent of `Sys`.

3 Transformation Definition

The conceptual model and its structure defined in Section 2 was introduced to define transformations of system models as shown in the left-hand side of Figure 1. A typical transformation step is the clustering of a group of sibling components within a container component, making them subcomponents of that container. Figure 2 shows the result of such a transformation clustering subcomponents ComponentB and SiblingSystem of component System in Figure 1 into a new System container. Besides introducing the new additional component System and making it a subcomponent of the original System root component, the transformation also requires changing

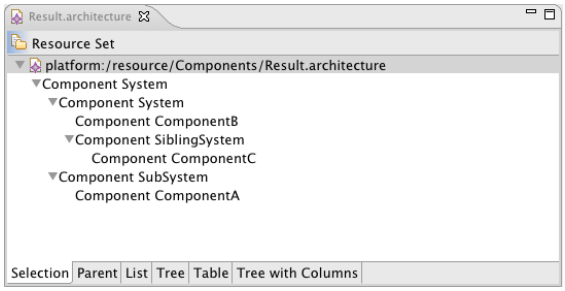


Figure 2: Example: Result of Clustering ComponentB and SiblingSystem

```
1  cluster(Pre,Group,Post) :-
2      Architecture(Pre,Pack,PreClass,PreAssoc),
3      Compound(PreComp,PreComps),union(OtherClass,[PreComp],PreClass),
4      subComp(PreSub,PreSubs),union(OtherAssoc,[PreSub],PreAssoc),
5      link(PreSubs,Group,PreRoot,OutSubs),
6      Compound(PreRootComp,PreRoot,Name),union([PreRootComp],Comps,PreComps),
7      subComp(NewSub,PostRoot,PreRoot),union([NewSub],OutSubs,InSubs),
8      Compound(PostRootComp,PostRoot,Name),union([PreRootComp,PostRootComp],Comps,PostComps),
9      link(PostSubs,Group,PostRoot,InSubs),
10     subComp(PostSub,PostSubs),union(OtherAssoc,[PostSub],PostAssoc),
11     Compound(PostComp,PostComps),union(OtherClass,[PostComp],PostClass),
12     Architecture(Post,Pack,PostClass,PostAssoc).
```

Figure 3: Cluster-Transformation: Rule for (De-)Constructing the Model

the supercomponent of ComponentB and SiblingSystem.

In a relational approach to model transformations, such a transformation is described as a relation between the model prior to the transformation (e.g., as given in the left-hand side of Figure 1) and the model after the transformation (e.g., as given in Figure 2). In this section, the basic principles of describing transformations as relations are described.

3.1 Transformations as Relations

In case of the clustering operation, the relation describing the transformation has the interface `cluster(Pre,Group,Post)` with parameter `Pre` for the model before the transformation, parameter `Post` for the model after the transformation, and parameter `Group` for the group of components of the model to be clustered. In the relational approach presented here, a transformation is basically described by breaking down the pre-model into its constituents and build up the post-model from those constituents using the relations from Section 2, potentially adding or removing elements and relations. With `Pre` taken from the conceptual domain described in Figure 1 and packaged in a single package *Architecture* with no sub-packages, it can be decomposed in contained classes (e.g., *Compound*) and associations (e.g., *subComp*) as shown in Figure 3,

```
1 link(Subs,[],Root,Subs).
2 link(InSubs,Group,Root,OutSubs) :−
3     subComp(SubRel,Sub,Root),union([Sub],Rest,Group),union([SubRel],Subs,InSubs),
4     link(Subs,Rest,Root,OutSubs).
```

Figure 4: Cluster-Transformation: Rule for (Un-)Linking SubComponents

lines 2 to 4.[5] In the same fashion, `Post` can be composed in lines 12 to 10. Lines 6 to 8 obtain the *Name* of the common super-component with entity *PreRoot* of the group (line 6), provide a newly created compound container component *PostRootComp* this *Name* and entity *PostRoot* (line 8), and make this *PreRoot* the super-component of *PostRoot* (line 7). Note that the relation is bidirectional: Besides clustering a group of siblings into a common container, it can also be used to uncluster the group of subcomponents contained in a common container.

Besides using the basic relations to construct and deconstruct models (and add or remove elements and relations, as shown in the next subsection), new relations can be defined to support a modular description of transformation, decomposing rules into sub-rules. E.g., in the `cluster` relation, the transformation can be decomposed into the addition of the new container component and the reallocation of the components to be clustered; for the latter, then a sub-relation `link` with corresponding rules is introduced, as shown in Figure 4. Note that `link` is effectively used in both directions in the `cluster` relation: In line 5, `link` is used to unlink subcomponents by removing the `subComp`-associations between *Group* elements and the original component *PreRoot* from *PreSubs* to obtain *OutSubs*; in line 9, `link` is used to link subcomponents by adding the `subComp`-associations between *Group* elements and the new component *PostRoot* to *InSubs* to obtain *PostSubs*.

3.2 Transformations as Rules

To define the transformation steps for (un)linking components and subcomponents, relation `link(InSubs,Group,Root,OutSubs)` is used, by making the set *OutSubs* of associations the reduction of set *InSubs* when removing all `subComp`-associations between elements from *Group* and *Root*. The (un)linking of a group depends on whether the group is empty or not. Therefore, in a declarative approach, two different – recursive – (un)link rules for those two cases are needed, each with the interface described above.

To define these rules as shown in Figure 4, the conceptual model and its structured representation introduced in Section 2 are used. Line 1 simply states that in case of an empty group the sets of associations are the same since no elements can be (un)linked. This case also handles the termination of the inductive rule definition. In case of a non-empty group, line 3 (un)links a *Sub* element from the *Group* – leaving a rest *Rest* – and *Root*, while line 4 repeats this (un)linking recursively for the *Rest* of the group. Note that this rule-based description allows to compose complex transformations by simple application of rules in the body of another rule (like `link` in

[5] For ease of reading, quotes required in Prolog for capital functor identifiers like *Architecture* or *Compound* are dropped.

`cluster`). In contrast, graphical specifications generally use additional forms of diagrams, e.g., state-transition diagrams. As shown in the following section, this direct combination of rules, however, is essential to simplify the formal verification of the correctness of transformations.

4 Verification

The relational and declarative approach introduced in the previous sections supports an easy transition to formal reasoning. In this section, the formalization of an EMF Ecore meta-model in constructive type theory is presented, as well as the straight-forward formalization of transformations. Based on these formalizations, the construction of formal correctness proof is demonstrated using the example of typical invariants. To support formal verification, the interactive theorem prover *Isabelle/HOL* [NPW02] is applied.

4.1 Meta-Model Formalization

Isabelle/HOL supports the form of (typed) terms used to represent the EMF models in the rule-based transformation process. Thus, the transition from the specifications used in Section 2 to *Isabelle/HOL* is straight-forward, as shown in the – syntactically slightly simplified – formalization of the meta-model of Figure 1:[6]

```
1 typedecl ids
2 typedecl string
3 datatype comp = Comp ids string, atom = Atom ids string
4 datatype subComp = SubComp ids ids
5 datatype cls = Comp comp set | Atom atom set
6 datatype asc = SubComp subComp set
7 datatype architecture = Architecture cls set asc set
```

After introducing – via `typedecl` – uninterpreted *ids* and *string* types for representing entities and string attributes in lines 1 and 2, the corresponding element (line 3), relation (line 4), class (line 5), association (line 6), and package (line 7) term types are introduced simply by providing – via `datatype` – constructor functions, using the same scheme as introduced in Subsection 2.1.[7] Based on these constructors and using the set operations provided by *Isabelle/HOL*, Prolog model terms can be directly translated, thus enabling the translation of transformations.

4.2 Transformation Formalization

Besides type terms, *Isabelle* also supports the definition of predicates in a rule-based fashion analogue to the Prolog-based rules in the transformation approach. To define the transformation relations in *Isabelle*, *inductive* definitions of predicates are used to allow recursive definitions. The non-recursive `cluster` relation of Section 3 is – trivially inductively – defined via:[8]

[6] *set* introduces a set type, `|` a variant type, `=>` a function type.

[7] The Compound and AtomicComponent element/class constructors are abbreviated to Comp and Atom, resp.

[8] Standard *Isabelle* notation is used, including `&`, `|`, and `--->` for conjunction, disjunction, and implication; `<=` and `:` for the subset and element relation; `?` for the existential quantor.

```
1  inductive cluster :: architecture => ids set => model => bool where
2      pre = (Architecture preclass preassoc) &
3      precomp = (Comp precomps) & otherclass Un {precomp} = preclass &
4      presub = (SubComp presubs) & otherassoc Un {presub} = preassoc &
5      (link presubs group preroot outsubs) &
6      prerootcomp = (Comp preroot name) & {prerootcomp} Un comps = precomps &
7      newsub = (SubComp postroot preroot) & {newsub} Un outsubs = insubs &
8      postrootcomp = (Comp postroot name) & {prerootcomp,postrootcomp} Un comps = postcomps &
9      (link postsubs group postroot insubs) &
10     postsub = (SubComp postsubs) & otherassoc Un {postsub} = postassoc &
11     postcomp = (Comp postcomps) & otherclass Un {postcomp} = postclass &
12     post = (Model postclass postassoc)
13         --> (cluster pre group post)
```

Obviously, again the transition from the specifications used in the previous sections to *Isabelle/HOL* is straight-forward: Line 2 to 12 directly correspond to line 2 to line 12 in Figure 3; in the former only a direct formalization with equality combined the constructors and set union is used, while the later uses (de)construction predicates. Line 13 of the former corresponds to line 1 of the later. Line 1 additionally defines the type of the predicate in *Isabelle/HOL* designated by "::". In a similar fashion, the specification of `link` can be directly translated:

```
1  inductive link :: subComp set => ids set => ids => subComp set => bool where
2      (link subs {} root subs)
3      (link subs rest root outsubs) --> (link ({subComp.SubComp sub root} Un subs) ({sub} Un rest) root outsubs)
```

4.3 Proof Construction

Using the formalization of the transformations introduced above, now correctness properties of the clustering operation can be defined. In the following, two conditions – one concerning class and one concerning association properties – are considered:

1. Each Compound element contained in the pre-model is also contained in the post model.
2. Each subComp relation between a component and some super-component in the pre-model has a counter-part in the post-model for the same component and some – potentially different – super-component.

The first property is formalized as `theorem` *keep_Comp_cluster*:

```
1  theorem keep_Comp_cluster:
2      (cluster pre group post) & pre = (Architecture preclass preassoc) & post = (Architecture postclass postassoc) &
3      preComp = (Comp preComps) & preAtom = (Atom preAtoms) & {preComp, preAtom} = preclass &
4      postComp = (Comp postComps) & postAtom = (Atom postAtoms) & {postComp, postAtom} = postclass &
5      (somecomp:preComps) --> (somecomp:postComps)
```

This theorem is straightforward to prove, requiring no induction but only case distinction. Therefore, the proof is mainly performed by `apply`ing *Isabelle*'s automatic proof tactics (e.g., *auto*, *clarify*, *clarsimp*), rendering the theorem (or lemma) applicable in further proof steps:

```
1  apply auto
2  apply (erule pushpull.cases)
3  apply clarify
```

```
4 apply (drule equalityD1)
5 apply (drule equalityD2)
6 apply (drule Un_sub_D)
7 apply (drule Un_sub_D)
8 apply clarsimp
```

Beside the case distinction (line 2), the proof requires three standard simplifications (lines 1, 3, 8) and four simple interactions deadline with equality and sub-set relation properties, where the latter could also be further automized by providing suitable rules.

The second, more challenging property is formalized as theorem *keep_subComp_cluster*:

```
1 theorem keep_subComp_cluster:
2   (cluster pre group post) & pre = (Architecture preclass preassoc) & post = (Architecture postclass postassoc) &
3   preSubComp = (SubComp preSubComps) & {preSubComp} = preassoc &
4   postSubComp = (SubComp postSubComps) & {postSubComp} = postassoc &
5   (? root. (SubComp some root):preSubComps) ——> (? root .(SubComp some root):postSubComps)
```

The proof script for theorem *keep_subComp_cluster* uses the same steps as before; however, since the corresponding super-component in a subComp-relation in the post-model is different whether the sub-component is in the group to be clustered or not, the proof requires one additional step – a lemma application – for distinction between these cases. To that end, corresponding lemmata are introduced and proved, e.g., *keep_link_group* to deal with the case on non-group elements. Since this distinction essentially affects `link`, these lemmata operate on the `link` relation:

```
1 lemma keep_link_group: (link pre group old lsubs) & (link post group new rsubs) ——> (lsubs <= rsubs & some:group)
2   ——> (SubComp some root):pre ——> (SubComp some root):post
```

Since these lemmata make use of the inductively defined relation `link`, induction must be used. However, besides suggesting the use of the induction principle on the definition of `link`, again the proof can performed fully automatic. These lemmata can be combined in a single lemma *keep_link* with a trivial proof:

```
1 lemma keep_link: (link pre group old lsubs) & (link post group new rsubs) ——> lsubs <= rsubs
2   ——> (? root. (SubComp some root):pre) ——> (? root. (SubComp some root):post)
```

In its proof, proven lemmata like *keep_link_group* can be applied in the form

```
1 apply (insert keep_link_group [of pre group old lsubs post new rsubs some])
```

The complete proof of theorem *keep_subComp_cluster* consists of the proof of the lemmata with 23 steps and 10 steps for the proof of the theorem itself with the resulting *keep_link* lemma.

5 Conclusion and Outlook

The PETE transformation framework – provided as an Eclipse PlugIn [Sch09] – supports the transformation of EMF Ecore models using a declarative relational style and allows a simple, precise, and modular specification of transformation relations on the problem- rather than the implementation-level. By including operational aspects, the relational declarative form of specification can be tuned to ensure an efficient execution. In the application to problem from the embedded software domain, the approach has demonstrated practical feasibility for medium real-world sized models (e.g, refactoring models consisting of more than 3000 elements and

more than 5000 relations within a few seconds). Furthermore, debugging on the level of the specification supports the construction of transformations.

The use of a declarative relational style of specifying transformations is an important asset for the formal verification of correctness conditions of these transformations: It allows the direct translation of the conceptual model as well as the transformation rules into a predicate-logical formalization. Since no indirections are introduced between the specifications on the execution and the verification level, the proof can be constructed following a natural argumentation. Using a verification tool like *Isabelle/HOL*, the verification process can be automized to a large extent.

While the previous sections have demonstrated the applicability of the approach, additional means of automation should be provided for a extensive application. This includes the mechanic translation of EMF Ecore models into the corresponding type definitions. Furthermore, the translation should include the definition of the basic manipulation predicates in the (de)constructor format to allow the 1:1 use of the executable specification of transformations in the verification. Additionally, general lemmata, tailor-made tactics, or using *ISAR* for more readable proof scripts should be provided to simplify proofs. Also, other property languages like OCL and pre/post schemata should be included, to circumvent the specification of property conditions on the level of predicate logics. Finally, the practicability of the verification approach requires the analysis of larger case studies.

Since the declarative relational style can also be used to support a search-based design-space exploration involving backtracking – e.g., when computing correct deployments in embedded systems – making test-based verification even more complex, simple formal verifiability of the correctness of such explorative transformations is especially helpful.

Bibliography

[GGL05] L. Grunske, L. Geiger, M. Lawley. A Graphical Specification of Model Transformations with Triple Graph Grammars. In Hartman and Kreische (eds.), *Model Driven Architecture*. LNCS 3748. Springer, 2005.

[GGL+06] H. Giese, S. Glesner, J. Leitner, W. Schfer, R. Wagner. Towards Verified Model Transformations. In *In Proceedings of MoDeVa workshop associated to MoDELS'06*. Pp. 78–93. 2006.

[KKS07] F. Klar, A. Königs, A. Schürr. Model Transformation in the Large. In *ESEC/FSE'07*. ACM Press, 2007.

[NPW02] T. Nipkow, L. C. Paulson, M. Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. Lecture Notes in Computer Science 2283. Springer, 2002.

[SBPM07] D. Steinberg, F. Budinsky, M. Paternostro, E. Merks. *EMF: Eclipse Modeling Framework*. Addison Wesley Professional, 2007. Second Edition.

[Sch08] B. Schätz. Formalization and Rule-Based Transformation of EMF Ecore-Based Models. In Dragan Gasevic (ed.), *Software Language Engineering*. LNCS. Springer, 2008.

[Sch09] B. Schätz. Prolog EMF Transformation Eclipse-PlugIn. `www4.in.tum.de/~schaetz/PETE`, 2009.

[Str08] M. Strecker. Modeling and Verifying Graph Transformations in Proof Assistants. *Electr. Notes Theor. Comput. Sci.* 203(1):135–148, 2008.

[VP04] D. Varro, A. Pataricza. Generic and meta-transformations for model transformation engineering. In Baar et al. (eds.), *UML 2004*. Springer, 2004. LNCS 3273.

Electronic Communications of the EASST
Volume of the Pre-proceedings of GT-VMT (2010)

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

Preserving constraints in horizontal model transformations

Paolo Bottoni, Andrew Fish[1], Francesco Parisi Presicce

15 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

[1] Partially funded by UK EPSRC grant EP/E011160: Visualisation with Euler Diagrams.

Preserving constraints in horizontal model transformations

Paolo Bottoni[1], **Andrew Fish**[2†], **Francesco Parisi Presicce**[1]

[1] Dipartimento di Informatica, "Sapienza" Università di Roma, Italy, [2]Computing, Mathematical and Information Sciences, University of Brighton, UK

Abstract: Graph rewriting is gaining credibility in the field of model transformation, and tools are increasingly used to specify transformation activities. However, their use is often limited by special features of the different graph transformation approaches, which might not be familiar to experts in the modeling domain. On the other hand, transformations for specific domains may require several constraints to be enforced on the results of the transformation. Preserving such constraints by manual definition of graph transformations can be a cumbersome and error-prone activity. In this paper we explore the problem of ensuring that possible violations of constraints following a transformation are repaired in a way coherent with the intended meaning of the transformation. In particular, we consider the use of transformation units within the DPO approach for intra-model transformations, where the modeling language is expressed via a type graph and graph conditions. We derive additional rules in a unit from a declarative rule expressing the principal objective of the transformation, so that the constraints set by the type graph and the graph conditions hold after the application of the unit. The approach is illustrated with reference to a diagrammatic reasoning system.

Keywords: DPO, automatic generation, model transformation

1 Introduction

Graph rewriting-based tools are increasingly used in the field of model transformation. However, their use is often limited by the special features of the different graph transformation approaches, which might not be familiar to experts in the modeling domain. On the other hand, transformations for specific domains may require constraints to be enforced on the results of the transformation. In this paper we explore the problem of ensuring that possible violations of constraints are managed in a way coherent with the intended meaning of the transformation.

We consider horizontal (or in-place) model transformations which destructively update a model expressed in a given language, for the case where the modeling language is expressed via a type graph and a set of graph conditions. In particular, we study transformations in reasoning processes deriving inferences via logical steps creating or deleting model elements.

While modelers are generally clear on what they want to achieve by defining a transformation, the evaluation of all of its consequences may be complex, and the definition of the implied preserving or enforcing actions cumbersome and error-prone.

We propose an approach to the automatic construction of transformation units achieving the effect of an intended model transformation while ensuring that all conditions are satisfied at the

† Partially funded by UK EPSRC grant EP/E011160: Visualisation with Euler Diagrams.

end of the unit if they held at its start. We consider transformations consisting of the creation or deletion of elements of a specific type, expressed as *principal* declarative rules. As their application may violate some conditions, they have to be applied in a proper (*condition preserving*) context, or (*condition enforcing*) repair actions have to be taken to restore the satisfaction of such conditions. Hence, additional rules are defined, derived from the principal one and the conditions to be enforced. The approach is illustrated with reference to a diagrammatic reasoning system.

In the rest of the paper, Section 2 discusses related work dealing with constraint preservation in graph transformation, and Section 3 introduces the relevant formal notions. Section 5 develops the proposed approach and Section 6 applies it to diagrammatic reasoning. Finally, Section 7 draws conclusions and points to possible future developments.

2 Related work

Rensink and Kuperus have exploited the notion of nested graphs to deal with the amalgamated application of rules to all matches of a rule. In [RK09], they define a language to specify nested graph formulae. A match can be found from a nested graph rule to a graph satisfying a formula, according to a given morphism, and the application of a composite rule ensues. Their approach is focused on avoiding control expressions when all the matches of a rule have to be applied, while we focus here on preserving constraints with reference to a single match.

Bottoni *et al.* have defined methods to extend single declarative rules for model transformation so that they comply with specific patterns defining consistency of interpretation in triple graphs [BGL08]. They define completions of single rules with respect to several patterns, while we are interested here in constructing several rules, navigating along different sets of constraints.

Taentzer *et al.* have proposed the management of inconsistencies among different viewpoints of a model in distributed graph rewriting. For example, the *resolve* strategy requires the definition of the right-hand sides of rules to be applied when the left-hand side identifying the inconsistency is matched [GMT99]. The detection of inconsistencies between rules representing different model transformations has been attacked by static analysis methods in [HHT02]. Similarly, Münch *et al.* have added *repair actions* to rules in case some post-conditions are violated by rule application [MSW00]. In all these cases, actions were modeled through single rules.

Habel and Pennemann [HP09] unify theories about application conditions from [EEHP06] and nested graph conditions from [Ren04], lifting them to high-level transformations. They transform rules to make them preserve or enforce both universal and existential conditions. Their approach leads to the generation of a single rule incorporating several application conditions derived from different conditions with reference to the possible matches of the rule on host graphs. In his dissertation [Pen09], Pennemann expands on the topic, also introducing programs with interfaces, analogous to transformation units, but allowing passing of matches.

In [OEP08], Orejas *et al.* define a logic of graph constraints to allow the use of constraints for language specification, and to provide rules for proving satisfaction of clausal forms.

The idea of introducing basic rules derived from entities and associations defined in a meta-model is exploited in [BQV06] to define constraints on the interactive composition of complex rules, by allowing their presence in the rule left or right-hand sides only in accordance with their roles in the meta-model, where only the abstract syntax is taken as a source of constraints.

Ehrig *et al.* describe a procedure, exploiting layers, which derives a grammar to generate (rather than transform) instances of the language defined by a meta-model with multiplicities [EKTW06]. Satisfaction of OCL constraints is checked a posteriori on a generated instance.

3 Background

For a graph $G = (V(G), E(G), s, t)$, $V(G)$ is the set of *nodes*, $E(G) \subset V(G) \times V(G)$ the set of *edges* and $s, t : E \to V$ the *source* and *target* functions. In a *type graph* $TG = (V_T, E_T, s^T, t^T)$, V_T and E_T are sets of node and edge types, while $s^T : E_T \to V_T$ and $t^T : E_T \to V_T$ define source and target node types for each edge type. G is typed on TG via a graph morphism $type\colon G \to TG$, where $type_V : V \to V_T$ and $type_E\colon E \to E_T$ preserve s^T and t^T, i.e. $type_V(s(e)) = s^T(type_E(e))$ and $type_V(t(e)) = t^T(type_E(e))$. $|V(G)|_t$ is the number of nodes of type $t \in V_T$ in G.

A DPO rule [EEPT06] consists of three graphs, called left- and right-hand side (L and R), and interface graph K. Two injective morphisms[1] $l\colon K \to L$ and $r\colon K \to R$ model the embedding of K (containing the elements preserved by the rule) in L and R. Figure 1 shows a DPO direct derivation diagram. Square (1) is a pushout (i.e. G is the union of L and D through their common elements in K), modeling the deletion of the elements of L not in K, while pushout (2) adds the new elements, i.e. those present in R but not in K. Figure 1 also illustrates the notion of *negative application condition* (NAC), as the association of a set of morphisms $n_i\colon L \to N_i$, also noted $NAC \xleftarrow{\vec{n}} L$, with a rule. A rule is applicable on G through a match $m\colon L \to G$ if there is no morphism $q_i\colon N_i \to G$, with N_i in NAC, commuting with m (i.e. $q_i \circ n_i = m$).

In the rest of the paper we exploit the partial order $\leq$ induced, up to isomorphisms, by monomorphisms on the set of graphs, i.e. $g_1 \leq g_2 \Leftrightarrow \exists m : g_1 \hookrightarrow g_2$.

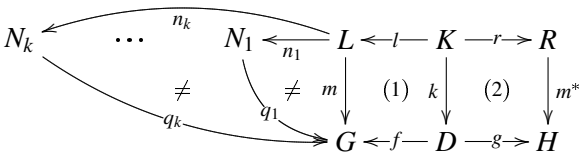


Figure 1: DPO Direct Derivation Diagram for rules with NAC.

Graph conditions allow the specification of models by forbidding the appearance of certain subgraphs, or by enforcing others to appear in given contexts. We use here a class of conditions $\mathscr{Q}$ similar to those in [HP09], where a condition over a graph A is either of the form `true` or of the form $\exists(a, q)$, with $a : A \to Q$ a morphism from A to some graph Q and q a condition over Q. Conditions are also obtained by using the Boolean connectives $\neg$ and $\vee$, and can be written in the form $\forall(a, q)$, equivalent to $\neg\exists(a, \neg q)$. We assume that all conditions in a set $\Theta \subset \mathscr{Q}$ differ for the a morphism, so that $(a_1, q_1), (a_2, q_2) \in \Theta \Rightarrow (A_1 \not\simeq A_2) \vee (Q_1 \not\simeq Q_2)$. We will also use the short forms $\exists(Q)$ for $\exists(a : \emptyset \to Q, true)$ and $\nexists(Q)$ for $\neg\exists(a : \emptyset \to Q, true)$.

In this paper we restrict ourselves to *positive* conditions of type $\forall(a : \emptyset \to Q, q)$, noted $\forall(Q, q)$ with q a disjunction of existential conditions, written as $q = \bigvee_{j \in J} q_j : Q \to W_j$, or of type $\exists(Q)$. Moreover, we admit simple *negative* conditions of the form $\nexists(Q)$. Note that, in this case, all the

[1] In this paper, when we speak of morphisms, we will always consider them injective.

conditions in Θ of the form $\exists(Q_i)$ can be collapsed into a single condition $\exists\overline{Q}$, where $\overline{Q}$ is the colimit of all Q_i on the diagram constructed with all pairwise maximal common subgraphs.

Definition 1 Given a graph G and a morphism $a : A \to X$, we say:

- A morphism $m : X \to G$ satisfies a condition C, ($m \models C$), *iff* one of the following holds:
  1. $C = true$.
  2. $C = \exists(Y)$ and $Y \leq X$.
  3. $C = \forall(X, \bigvee_{j\in J} q_j : X \to W_j)$ and $\exists m_j : m(X) \to W_j$ s.t. $q_j = m_j \circ m$ for some q_j.
  4. $C = \nexists(Y)$ and $Y \not\leq X$.
  5. $C = C_1 \vee C_2$ and $m \models C_1$ or $m \models C_2$.
- A graph G *satisfies* C ($G \models C$), *iff* one of the following holds:
  1. $C = true$.
  2. $C = \exists(Y)$ and there exists $m : Y \to G$ s.t. $m \models C$.
  3. $C = \forall(X, q)$ and for each $m : X \to G$, $m \models C$.
  4. $C = \nexists(Y)$ and there is no morphism $m : Y \to G$.
  5. $C = C_1 \vee C_2$ and $G \models C_1$ or $G \models C_2$.

We say that a graph G typed on TG is a *model* for (Θ), noted $G \models \Theta$, if $\forall C_i \in \Theta$, $G \models C_i$. In the rest of the paper, we assume we are dealing with a consistent set of conditions, admitting only models which are finite non-empty graphs. In particular, we consider simple graphs, with no two instances of the same edge type between two nodes.

Transformation units control rule application through control words over rule names [KKS97]. We set them here in the framework of the DPO approach for typed graphs with NACs, with: 1) $\mathscr{G}$ the class of typed graphs; 2) $\mathscr{R}$ the class of DPO rules on typed graphs with NACs; 3) $\Longrightarrow$ the derivation relation for the DPO approach; 4) $\mathscr{E}$ a class of graph expressions (here defined by type graphs and graph conditions), where the *semantics* of an expression e is a subclass $sem(e) \subset \mathscr{G}$; 5) $\mathscr{W}$ a class of control words over identifiers of rules in $\mathscr{R}$ built on a grammar allowing single rules, the sequential construct ';', the iteration construct w^*, with $w \in \mathscr{W}$, the alternative choice '|'. A transformation unit is a construct $TU = (e_1, e_2, P, imp, w)$, with $e_1, e_2 \in \mathscr{E}$ initial and terminal graph class expressions, $P \subset \mathscr{R}$ a set of DPO rules, imp a set of references to other, *imported*, units, whose rules can be used in the current one, and $w \in \mathscr{W}$ a control word enabling rules from P, and units from imp, to be applied. Transformation units have a transactional behaviour, i.e. a unit succeeds *iff* it can be executed according to the control condition; it fails otherwise. The semantics of a TU is the set $sem(TU) = \{(g_1, g_2) \mid g_1 \in sem(e_1), g_2 \in sem(e_2), g_1 \overset{TU\downarrow}{\Longrightarrow} g_2\}$, where $\downarrow$ indicates successful termination.

4 A Running Example: Spider Diagrams

Spider Diagrams are a reasoning system based on Euler diagrams. Several variants exist under the same name, differing in the allowed syntax and even the assigned semantics [HMT+01]. We

consider a simplified version, based on Venn, rather than Euler, diagrams and omitting shading and strands. We first provide an indication of the concrete syntax of the diagrams and give an informal indication of the semantics. Then we propose a graph-based abstract model for them, called *Spider Graphs*, which differs from the usual algebraic abstract models used, and is in fact slightly closer to the concrete model than usual, even modelling spider's feet.

Let $C = \{C_1, \ldots, C_n\}$ be a collection of simple closed curves in the plane with finitely many points of intersection between curves. A *zone* is a region of the form $X_1 \cap \cdots \cap X_n$, where $X_i \in \{int(C_i), ext(C_i)\}$, the interior of C_i or the exterior of C_i, for $i \in \{1, \ldots, n\}$. If each of the 2^n possible zones of C are non-empty and connected then C is a *Venn diagram* (see [Rus97] for more details). Each zone z defines a unique partition of the set C, according to whether z is *inside* or *outside* a curve. Two zones are called *twins* if their inside and outside relations are switched for exactly one curve. In this paper, a *Spider Diagram* is a Venn diagram whose curves are labelled, together with extra syntax called *spiders*, which are trees whose vertices (called *feet*) are placed in unique zones. The set of zones containing a spider's feet is called its *habitat*. Special arcs, called *ties*, can be drawn between feet of different spiders in the same zone.

Intuitively, each curve represents a given set (indicated by the label) and each zone represents some set intersection. A spider indicates the existence of an element within the set determined by its habitat, whilst a tie between a pair of feet of different spiders within a zone indicates equality of elements, if both spiders represent an element in the set represented by the zone.

The left hand side of Figure 2 shows an example of a Spider Diagram, with two curves $\{A, B\}$ and four zones described by $\{(\{A\},\{B\}), (\emptyset,\{A,B\}), (\{B\},\{A\}), (\{A,B\},\emptyset)\}$. Here, these zones are the four minimal region of the plane determined by the curves; for example, the zone described by $(\{A\},\{B\})$ is the region $int(A) \cap ext(B)$ which is inside A but outside B. The habitat of spider s is the set of zones $\{(\{A\},\{B\}), (\{A,B\},\emptyset)\}$, while that of t is the singleton $\{(\{A,B\},\emptyset)\}$. Informally, the diagram semantics is: there are two sets A and B, there exists an element named s in A and an element named t in $A \cap B$. Moreover, if s is in $A \cap B$ then $s = t$.

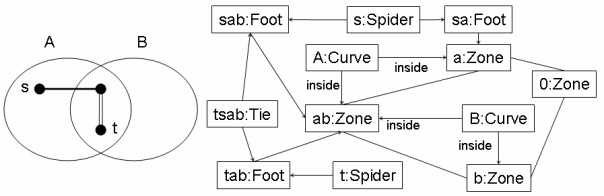


Figure 2: A Spider Diagram on the left, with the corresponding Spider Graph on the right.

We provide here an abstract graph-based model of a Spider Diagram, called a *Spider Graph*, which does not take into account the concrete geometry of the diagram. Since we are interested here only in syntactic aspects of the diagrams, we do not consider the labeling of the curves. We obtain the type graph shown on the left of Figure 3, where nodes represent the diagram elements *Curve*, *Foot*, *Spider* and *Zone*, and edges represent relations between them. In particular, a *twin* edge indicates that two zones are twins w.r.t. some curve and an inside/outside edge indicates whether a curve contains/excludes a zone, respectively.

On the right of Figure 2 the Spider Graph associated with the Spider diagram on the left is

shown. The names of the nodes show the correspondence with the objects in the diagram. We have two curve nodes in each possible relation with four zones[2]. For ease of reading, the zone nodes are given names consisting of a list of the lower case letters corresponding to the upper case letters used as names of the curves the zones are inside, and we use O for the name of the node corresponding to the zone outside all curves in the diagram. Zone node pairs ab and b, and O and a are twinned due to curve A, whilst ab and a, and O and b are twinned due to curve B.

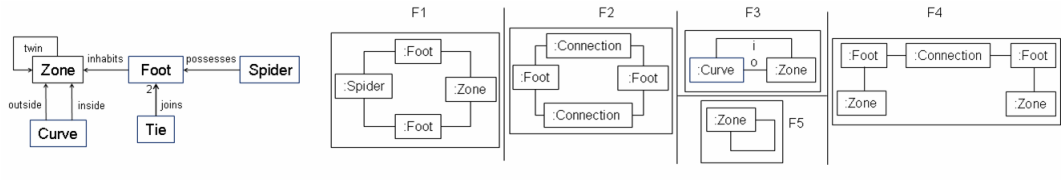


Figure 3: The type graph (left) and negative conditions (right) for Spider Graphs.

We now present the conditions completing the definition of the class of Spider Graphs. The right hand side of Figure 3 shows a set of conditions of the form $\nexists Q$, presented as forbidden graphs. They prevent duplication or inconsistency of information and state the uniqueness of relations between zones and curves. Moreover, we assume the existence of all negative conditions forcing the graphs to be simple. We omit the direction of edges and their labels, when understood from the type graph, and use the abbreviations `i` and `o` for the inside/outside case.

The remaining conditions force the existence of a partition of the set of curves for all zones, and require the existence of suitable contexts for zones and feet. We present them adopting a visual syntax where a condition $\exists(a : A \to Q, q)$ is represented by a box, separated into two parts by a horizontal line, with the top part containing a depiction of the morphism a and the bottom part containing a box depicting the condition q on Q. An empty bottom box corresponds to `true`. Each condition box has an external tab containing either quantifier information or the boolean connective $\vee, \wedge$ or $\neg$. As we use conditions with $A = \emptyset$, we only present Q and we do not repeat Q in the depiction of q. Numbers are used to indicate identification in the morphisms, while nodes that are not numbered indicate a hidden existential quantification, as usual. Edges between identified nodes are also assumed to be identified in the morphisms.

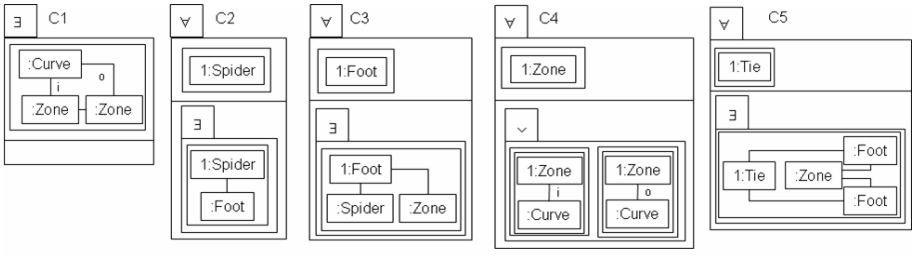


Figure 4: Conditions on single elements.

[2] To keep the graph simple, we have omitted the *outside* edges, which are complementary to the *inside* ones.

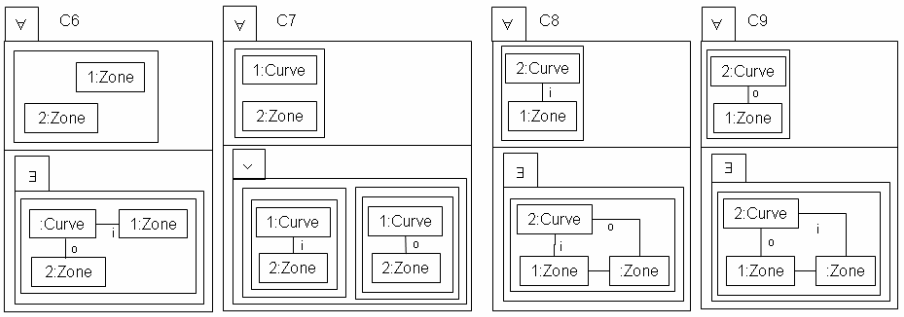


Figure 5: Conditions on pairs of elements.

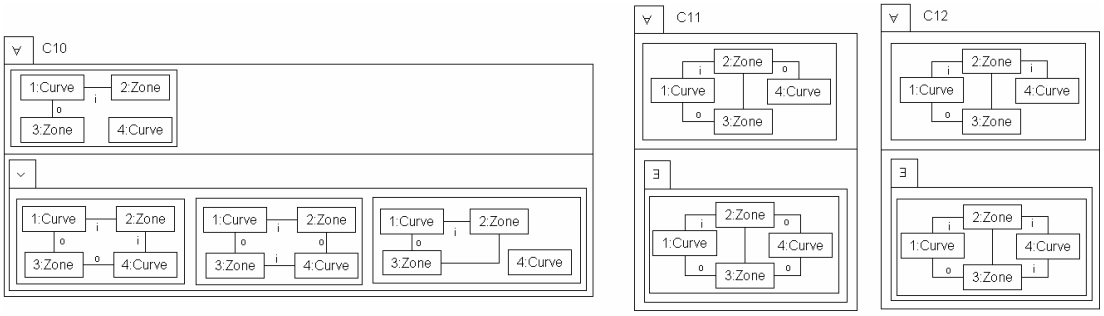


Figure 6: Conditions on existence and uniqueness of twins.

The class of Spider Graphs is the intersection of the languages defined by the type graph and the negative conditions of Figure 3, and the positive conditions in Figures 4 to 6.

Reasoning rules are derived on top of the algebraic abstract models for Spider diagrams. These are syntactic transformations whose application corresponds to logical deduction, according to the semantics. They are usually specified by complex algorithmic procedures, during which the intermediate diagrams may not be logical consequences of the premise diagram, with pre and post conditions taking into account the stated semantics of the diagram. For instance a rule to add a new curve must split every zone into two zones, one inside and one outside each existing zone, as well as duplicating spider's feet in zones. Whereas the first effect derives from the syntactical conditions, the second is a semantic aspect.

5 Condition preserving rules

We discuss the derivation of condition-preserving transformation units TU_g^t and TU_d^t in the case of generation or deletion, respectively, of an element of type t. The initial and terminal graph class expressions e_1 and e_2 for both of these units define the class of graphs typed on TG and satisfying Θ. TU_g^t will be associated with the execution of[3] $r : \emptyset \leftarrow \emptyset \rightarrow \boxed{t}$, where $+$ indicates the

[3] Here and in the rest of the paper, $\boxed{t}$ denotes the graph consisting of a single node of type t.

pushout along the empty subgraph, while TU^t_d with $r : \boxed{t} \leftarrow \emptyset \rightarrow \emptyset$. The units will be constructed so that given a graph $G \in sem(e_1)$, for $G \overset{TU^t_g\downarrow}{\Longrightarrow} H$, $(G,H) \in sem(TU^t_g)$, $G \leq G + \boxed{t} \leq H$ and $(H,G) \in sem(TU^t_d)$. For $G \overset{TU^t_d\downarrow}{\Longrightarrow} H$, we will have $(G,H) \in sem(TU^t_d)$, and $H + \boxed{t} \leq G$.

Note that in general $G + \boxed{t} \not\models \Theta$, but $G + \boxed{t} \models \Theta'$ for some $\Theta' \subset \Theta$. Hence, we admit that some conditions may not be satisfied at intermediate steps of the unit application, and define an *operational* class in which to perform transformations. Graphs in this class satisfy a subset of the graph conditions and may be typed on some TG' with additional types and edges w.r.t. TG. In particular, we use here the subset Θ' containing $\exists(\overline{Q})$ and all the conditions $\nexists(Q_i)$ in Θ.

Before presenting the algorithms, we give their rationale, starting with the case[4] $r : \emptyset \rightarrow \boxed{t}$. We only have to consider universal and negative existential conditions, as positive existential conditions cannot be violated by adding an element. However, adding $\boxed{t}$ produces a graph $G + \boxed{t}$ which may not satisfy Θ in two ways: either it contains a forbidden subgraph, or it provides a new match for the premise of a universal condition but fails to satisfy the conclusion.

To solve the first problem, given a $r : L \rightarrow R$ in TU^t_g (including $r : \emptyset \rightarrow \boxed{t}$), for each condition $\nexists(X) \in \Theta$, the function $genNAC(r,X)$ adds to r the set of NACs formed according to the construction on the left of Figure 7. Here M_j is a maximal common subgraphs of X and R, M'_j is a maximal common subgraph of M_j and L, and all the squares are pushouts, i.e., $M_j \rightarrow X_j \rightarrow X$ is the pushout complement for $M_j \rightarrow R \rightarrow X$, $L \rightarrow X'_j \leftarrow X_j$ is the pushout for $L \leftarrow M'_j \rightarrow X_j$, and the square with L, R, M_j and M'_j also forms a pushout. The set of NACs contains all the morphisms $n'_j : L \rightarrow X'_j$ preserving the image of L in X. This prevents the application of the rule on a match which could create the forbidden subgraph X (see [HHT96]).

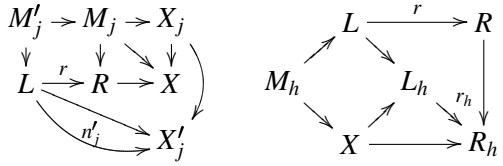


Figure 7: Constructing NAC (left) and incorporating available context (right).

To solve the second problem, given a (universal) condition $C = \forall(Q, \bigvee_{j\in J} q_j : Q \rightarrow W_j)$, the function $genUniRules(C)$ produces the set of rules $R(C)$ where each rule has the form $NAC(C) \overset{\vec{n}}{\leftarrow} Q \overset{r_{C,j}}{\rightarrow} W_j$. TU^t_g will contain an alternative choice among these rules, produced by the function $alt(R(C))$. In order to prevent these rules from being applied indefinitely in case of iteration on the choice, the set $NAC(C)$ contains a copy of each W_j to ensure that the same match is not reused twice for generating some conclusion. Intuitively, these rules will adjust the relations of the newly added element w.r.t. the contexts defined in their premises. Howevever, several aspects have to be taken into account. For example, consider conditions $C2$ in Figure 4 and suppose we want to add a *Spider*. Then, the derived rule will have to create a *Foot* (condition $C2$), but this will require a *Zone* (condition $C3$), which will require a *Curve* (condition $C4$), hence other additional *Zones* (conditions $C8$ and $C9$), with several relations to other curves and zones (conditions $C10-C12$). On the other hand, a *Zone* for a *Foot* is already guaranteed to be present

[4] Where not needed, we will omit K.

by $C1$, so that one can reuse existing context to satisfy this. To deal with such cases, given a rule $r : L \to R$ and a context X to be reused (more on this later), the function $reuseContext(r,X)$ produces a collection of rules of the form $r_h : L_h \to R_h$ according to the construction on the right of Figure 7. Here, $L \to L_h \leftarrow X$ is the pushout along a maximal common subgraph M_h of L and X and $X \to R_h \leftarrow R$ is the pushout of $X \leftarrow M_h \to R$.

In general, one wants to obtain a TU_g^t which, after applying $r : \emptyset \to \boxed{t}$ to G, proceeds through the following abstract steps, so that context is progressively constructed for the next step.

1. define all edges between the added node and existing nodes of G as required by conditions;
2. generate new nodes as required by the conditions;
3. generate all edges for the new nodes, as required by the conditions.

For example, when adding a `Curve`, one will have to: 1) define relations between new curve and existing zones; 2) create new zones, while defining relations with the new curve; 3a) establish relations between new zones and existing curves; 3b) establish relations between zones.

Two things have to be considered. In general, satisfaction of $\forall(Q,q)$ requires iterating through all possible matches for Q. However, when Q consists of just one node, no iteration is necessary, and if Q is the graph $\boxed{t}$, the derived rule has to be applied only to the newly added node, as it is already satisfied for the nodes of type t which were in G originally. Hence, we extend TG to admit a special type of loop edge: the first rule is changed to $r : \emptyset \to \boxed{t}^\dagger$, where $\boxed{t}^\dagger$ designates a node with a marker loop. For a rule[5] $r : L \to R$, the function $mark(r)$ produces a set $P_r^\dagger = \{r_h^\dagger : L_h^\dagger \to R_h^\dagger \mid h : \boxed{t} \to L\}$ where $L_h^\dagger$ and $R_h^\dagger$ are obtained by adding the loop to the images $h(\boxed{t})$ and $r \circ h(\boxed{t})$, the immersions $m_h : L \hookrightarrow L_h^\dagger$ and $m'_h : R \hookrightarrow R_h^\dagger$ preserve the images of $\boxed{t}$ under h, and $r_h^\dagger$ is the unique morphism s.t. $L_h^\dagger \xrightarrow{r_h^\dagger} R_h^\dagger \xleftarrow{m'_h} R$ is the pushout of $R \xleftarrow{r} L \xrightarrow{m_h} L_h^\dagger$. TU_g^t will apply r or rules from $P_r^\dagger$ dependent upon the situation. The rule $delLoop : \boxed{t}^\dagger \to \boxed{t}$ will conclude TU_g^t deleting the loop.

Moreover, as seen from the examples above, the application of some rules can require the creation of new nodes, if they cannot be provided by the context, and so conditions relative to the new nodes have to be satisfied. This potentially creates a case in which an infinite recursion might start. To avoid this, we study the relations between types for which conditions are mutually recursive. In our example, one such pair consists of `Curve` and `Zone`. Indeed, the generation of a curve implies the generation of a collection of zones, whilst the generation of a zone can imply the generation of a single curve and of the collection of zones related to the new curve. Again, we need to distinguish between cases in which context, enriched with the new node which has started the process, has to be reused, and those in which a new node is needed to provide the correct context. Definition 2 provides notation to assist us.

Definition 2 Let $t \in V_T$ be a type and $Q(t) \subset \Theta$ the set of conditions of the form $Op(a : A \to Q,q)$, for $Op \in \{\exists,\forall,\nexists\}$ s.t. $\boxed{t} \leq Q$ (i.e. a node of type t appears in Q). $\{Q_\exists(t),Q_\forall(t),Q_\nexists(t)\}$ is a partition of $Q(t)$ into *existential*[6], *universal and negative existential* conditions for t, respectively.

[5] For each function operating on rules or types we overload the symbol to accept as argument sets.
[6] Note that $Q_\exists(t) = \{\exists(\overline{Q})\}$ if $\boxed{t} \leq \overline{Q}$, and $Q_\exists(t) = \emptyset$ otherwise.

$V_T^{\exists} = \{t \mid \boxed{t} \leq \overline{Q}\}$ is the set of *existentially quantified types*. A partial order $\leq_C$ is induced on $Q_\forall(t)$ by $(C_1 <_C C_2) \Leftrightarrow ((A_1 < A_2) \vee ((A_1 \simeq A_2) \wedge (Q_1 < Q_2)))$. $DAG(t)$ is the directed acyclic graph induced on $Q(t)$ by $<_C$, where (q_1, q_2) is an edge of $DAG(t)$ *iff* $q_1 <_C q_2 \wedge \nexists q_x$ s.t. $q_1 <_C q_x, q_x <_C q_2$. We call $Min(t)$ the set of minimal models for $\Theta \cup \{\exists(\overline{Q} + \boxed{t})\}$ for $t \in V_T \setminus V_T^{\exists}$ and $MIN(S)$ the set of minimal models for $\Theta \cup \bigcup_{t \in S \subset V_T} \{\exists(\overline{Q} + \boxed{t}\}$.

For each condition $C \in Q_\forall(t)$ the rules in $genUniRules(C)$ will be applied in an order established by a function $visit(DAG(t))$ which starts from initial nodes and proceeds from a join node only after all its incoming paths have been visited. In this way, progressively increasing contexts will have been produced, possibly providing new matches for the following ones.

In order to follow the abstract steps discussed above, for a type t we organize the rules derived from $Q_\forall(t)$ into layers: $LAYER_1(t)$ contains rules which only add edges touching nodes of type t, $LAYER_2(t)$ contains rules which add at least one node (of any type) in a non-empty context (and possibly edges of any type), whilst $LAYER_3(t)$ contains rules which do not create nodes but add edges of any type, but with at least one edge between instances of some type other than t.

The sets $Min(t)$ provide context which is certainly present if a unit for the addition of an element of type t has already been applied, while $\overline{Q}$ is guaranteed to be always present. Hence, *reuseContext* will be invoked with parameter X equal to $\overline{Q}$ or $Min(t)$, dependent on the situation.

Moreover, if an element of type t' is created as a consequence of the generation of $\boxed{t}$, rules derived from the visit of $DAG(t')$ have also to be applied, in the context provided by the already applied rules. In order to avoid infinite recursion, we introduce a notion of *domination* and say that $DAG(t)$ dominates $DAG(t')$ if $DAG(t') < DAG(t)$. Figure 8 shows the *DAG*s for the example introduced in Section 4. It is easy to see that $DAG(Zone)$ dominates $DAG(Curve)$ so that the construction of TU_g^{curve} should recursively be invoked and the rules from $Q_\forall(curve)$ should be used twice, creating new zones. To avoid this problem, if $DAG(t') \leq DAG(t)$, then the rules from $DAG(t')$ outside the recursion will be processed via *reuseContext*, with $X = MIN(t')$. We also use a function $createdType(r)$ returning the set of types produced by r, i.e. in $V_T(R) \setminus V(L)$.

The last aspect to consider is that there might be conditions which have the same conclusions from different premises, but such that the premises of both are included in the conclusion. Such is the case of conditions $C8$ and $C9$ in Section 4. Hence, the function $sameConcl(RS)$ processes a set of rules RS to add NACs to such pair of rules.

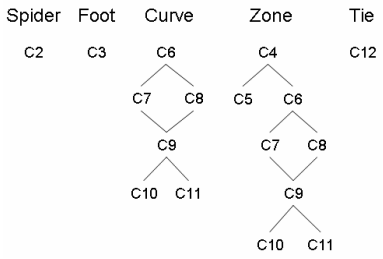


Figure 8: The DAGs for Spider Graphs.

We are now ready to present the algorithm *CreateGenUnit(t)*, deriving TU_G^t exploiting rules generated from the universal conditions, organizing them in accordance with the ordering and

layering, so that rules are applied only when their context for application is ready, and adding NACs to prevent violation of negative existential conditions.

`Algorithm` $CreateGenUnit(t:\texttt{type}):\texttt{TU}$

initialize global $UNIT$ with $r_t^{\dagger}:\emptyset\to\boxed{t}^{\dagger}$;
foreach condition $C=\forall(Q,q)\in\Theta$ **do** { $R(C)=sameConcl(genUniRules(C))$; }
return $RecursiveGen(t,\emptyset,false)$;

`Algorithm` $RecursiveGen(t:\texttt{type},\ S:\texttt{setOfTypes},inner:\texttt{boolean}):\texttt{TU}$

$path=visit(DAG(t))$; $X=\emptyset$; $aux=\emptyset$;
if $isEmpty(S)$ **then** { **if** $t\in V_T\setminus V_T^{\exists}$ **then** { $X=\overline{Q}$; } } **else** { $X=MIN(S)$; }
foreach condition $C=\forall(Q,\bigvee_{j\in J}q_j:Q\to W_j)\in path$ **do** {
  foreach $k\in\{1,\ldots,3\}$ **do** {
    foreach $t'\in S$ { **if** $(dominates(DAG(t),DAG(t')))$ { $aux=aux\cup\{t'\}$; } };
    if $!isEmpty(aux)$ **then** { $X=MIN(aux)$ }; $single=\emptyset$; $nosingle=\emptyset$;
    foreach rule $r_{C,h}=NAC\xleftarrow{n}L\to R\in R(C)\cap LAYER_k(t)$ **do** {
      if $|\,V(L)\,|=1$ **then** { $single=single\cup\{r_{C,h}\}$; } **else** { $nosingle=nosingle\cup\{r_{C,h}\}$; };
      if$(inner)$ **then** { $UNIT=concat(UNIT,alt(reuseContext(single,X)))$;
        $UNIT=concat(UNIT,(alt(reuseContext(nosingle,X)))^*)$;
      } **else** { $UNIT=concat(UNIT,alt(mark(reuseContext(single,X))))$;
        $UNIT=concat(UNIT,(alt(mark(reuseContext(nosingle,X)))))^*)$; };
      if $(k==2)$ **then** { **foreach** $t'\in createdType(r_{C,h})$ **do** {
        $UNIT=concat(UNIT,RecursiveGen(t',S\cup\{t\},true))$; } } };
foreach rule $r:L\to R$ in $UNIT$ **do** {
  foreach condition $C=\nexists(X)\in\Theta$ **do** { **replace** r with $genNAC(r,X)$; } };
$UNIT=concat(UNIT,delLoop)$;
return $UNIT$

Theorem 1 *A call $CreateGenUnit(t,\emptyset)$: 1) terminates, and 2) produces a unit TU_g^t s.t. given a graph G typed on TG, if $G\models\Theta$ then $\forall H\ [(G\overset{TU_g^t\downarrow}{\Longrightarrow}H)$ implies $(H\models\Theta\cup\{\exists(G+\boxed{t})\})]$.*

Proof. (**Sketch**) 1) The first nested loop performs a finite number of iterations on conditions, layers and rules. The recursion on *recursiveGen* terminates since the set S increases in size on each call. The final iteration to add NACs occurs on a finite number of conditions and rules.

2) If the first rule is applicable, then the application of $TU_G(t)$ terminates on each finite graph G s.t. $G\models\Theta$. Indeed, the NACs prevent repeated applications of a rule on identical matches, and even if new matches can be created, the layering prevents infinite repetition of the execution of a rule. Moreover, the application of *reuseContext* avoids arbitrary generation of new elements. If a graph H is obtained, then $H\models\exists(G+\boxed{t})$, as only increasing rules have been applied. Suppose now that $H\not\models\Theta$. Then either: 1) $H\not\models C_i$ for some C_i in some $Q_{\nexists}(t)$, but this is impossible as this is prevented by the use of *genNAC*; or 2) $H\not\models\exists\overline{Q}$, but this is impossible as $\overline{Q}\le G\le G+\boxed{t}\le H$; or 3) $H\not\models C_i$ for some C_i in some $Q_{\forall}(t)$, but this is impossible as all the rules are derived from some $Q_{\forall}(t)$ and all matches for their premises have been considered. □

Due to lack of space, we just present the main ideas concerning the generation of TU_d^t, i.e. of a unit for deleting an element of type t. Now, the modeler can mark a node to indicate which instance to delete. Hence, TU_d^t will start by applying a rule $\boxed{t}^\dagger \xleftarrow{n} \boxed{t} \xleftarrow{l} \boxed{t} \xrightarrow{r} \boxed{t}^\dagger$ and finish with $\boxed{t}^\dagger \xleftarrow{l} \emptyset \xrightarrow{r} \emptyset$. The logic of the unit proceeds in reverse w.r.t. element generation. Hence, it will delete edges first, starting with those not directly connected to the element, but to its context, then additional elements relative to the marked instance, and finally all the remaining edges for the element. Given a condition C, a function $delUniRule(C)$ will produce a set $D(C)$ of deleting rules, inverting the role of premise and conclusion w.r.t $genUniRule(C)$. Hence, each rule in $D(C)$ has the form $W_j \xleftarrow{l_{C,j}} Q \xrightarrow{r_{C,j}} Q$. However, care must be taken that the removal of t does not destroy $\overline{Q}$. Hence, for a rule $L \leftarrow R \rightarrow R \in D(C)$, the function $preserveMinimal(r)$ produces a collection of rules of the form $L +_{M_i} \overline{Q} \xleftarrow{n_i} R +_{M_i} \overline{Q} \xrightarrow{n_i} R +_{M_i} \overline{Q}$, where $+_{M_i}$ denotes the pushout along a maximal common subgraph M_i of R and $\overline{Q}$. Hence, no element can be deleted if this would disrupt the minimal model. Moreover, as $t^\dagger$ is deleted only when it has no more defining context, the conditions in $Q_\forall(t)$ are either vacuously satisfied (if $t \in V_T^\exists$ and this was the only instance of t and left in G, the unit will fail), or satisfied by the remaining instances.

6 Application to Spider Diagrams

We now consider the application of *CreateGenUnit* on Spider Graphs as defined in Section 4.

Firstly, considering the addition of a *Curve*, the conditions in $Q_\forall(Curve)$ are ordered according to $DAG(Curve)$ in Figure8, while $Q_\exists(Curve) = \{C1\}$, and $Q_\nexists(Curve) = \{F3\}$ together with the conditions preventing duplication of edges of the same type between a curve and a zone. As for layering, we have: $LAYER_1(Curve)$ contains rules generated from $C7$, $C11$, $C12$ and from the first two graphs in the bottom box of $C10$ since these only add edges incident with nodes of type *Curve*. $LAYER_2(Curve)$ contains rules generated from $C8$,$C9$ which add *Zone* nodes, whilst $LAYER_3(Curve)$ consists of the rule generated from the last graph in the bottom box in $C10$ which adds an edge between nodes of type *Zone*. Note that for $C8$ and $C9$, $sameConcl(R(Curve))$ will produce NACs preventing the generation of an infinite number of nodes.

Figure 9 shows a version of the rules in $R(C10)$, derived from applying $genUniRules(C10)$, with one choice of marking for the curve. According to the construction, the set $NAC(C10)$, presented at the bottom left of Figure 9, contains a *NAC* for each possible right-hand side, preventing reuse of the same match for rules from $R(C10)$.

Using the same rule naming scheme as in Figure 9, and the initial rule $r_{curve} : \emptyset \rightarrow \boxed{Curve}$, the algorithm produces TU_g^{Curve} of the form:

$$TU_g^{Curve} = r_{curve}^\dagger;\ (r7.1^\dagger \mid r7.2^\dagger)^*;\ (r10.1^\dagger \mid r10.2^\dagger)^*;\ r11^{\dagger *};\ r12^{\dagger *};\ r8^{\dagger *};\ r9^{\dagger *};\ (\overline{r4.1} \mid \overline{r4.2});\ \overline{r6}^*;\ (\overline{r7.1}^* \mid \overline{r7.2}^*);(\overline{r10.1}^* \mid \overline{r10.2}^* \mid \overline{r10.3}^*);\ \overline{r11}^*;\ \overline{r12}^*;\ r10.3^{\dagger *};$$

The internal sequence of iterations derives from the fact that some *Zone* should be created by either a rule in $r11^*$ or in $r12^*$, so that the unit for zones is activated. However, these reuse context from the unit for the curve, so that they reuse context and do not create new elements, as indicated by their presence in an $\overline{r}$ version. On the other hand, the rules related to the curve use the † version, to refer to the curve for which the unit is started. Note that rule 10.3 appears in

layer 1 for zones and layer 2 for curves.

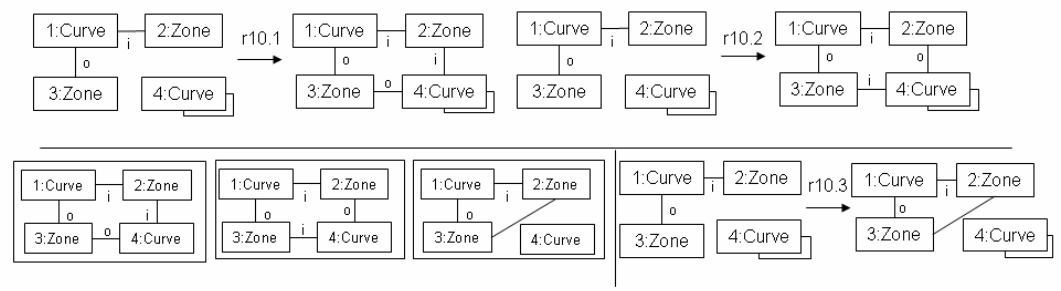


Figure 9: A marked version of the 3 rules derived from condition C10 and the non-marked NAC.

For a Spider, we have $Q_{\exists}(Spider) = \emptyset$, $Q_{\forall}(Spider) = \{(C2)\}$, generating a rule in layer 2. While the creation of a `Spider` requires the creation of a `Foot`, the `Zone` will be taken from the context, due to its presence in $\overline{Q}$, so that it has been incorporated by the application of *reuseContext*. Insertion of a `Foot` will instead require the creation of a new `Spider`, if none exists, or its reuse if one had already been created. However, such a creation will fail if the spider has already a foot in each existing zone.

Deletion of a curve is performed by first removing the twin edges between zones attached to the marked curve, then removing all edges from all other curves to these zones, then all zones attached with an *inside* or *outside* edge to the curve to be removed, then all remaining connections from the marked `Curve` node to be deleted, and finally the marked node itself. In a similar way, removal of a spider will be preceded by removal of all its feet and their attachments to zones.

7 Conclusions

We have provided a methodology for the automatic derivation of transformation units from a principal rule via algorithms that iteratively add preparatory rules to the unit for deleting rules and that add restorative rules to the unit for increasing rules. As a result, membership in the model language is ensured before and after the application of the unit, but not necessarily throughout the unit. The derivation of the units exploits a rule layering approach, with the rules within the unit generated from graph conditions taking into account the rule application context.

The automatic production of the rules needed to reassemble a syntactically correct diagram simplifies the specification of diagrammatic inference rules and supports therefore the development and comparison of syntactic and semantic variations of the systems.

This would be useful also in other domains. For instance, model refactoring often involves the elimination of elements, or the creation of suitable contexts for their insertion. One example is the elimination of a composite state in a Statechart which requires the elimination of all of its internal states. Then, given a set of conditions stating that each state must be contained within a composite state, the construction in Section 5 could be applied to generate transformation units to be recursively invoked to visit the nesting tree. Another refactoring example is that of moving a method. This requires placing it in a different class and redirecting all its invocations, as well

as the messages which may originate from its invocation, to its new location. Our construction can thus be used to manage the identification of the arcs related to such a method.

Of course, semantic considerations play a greater role than simple syntactic constraints. However, the constructed rules may provide a basis to be extended with additional context and consequences. For example, the specification of a transformation via pre- and post-conditions can be used to integrate syntactic rules with specific side effects. In this sense, this construction provides more flexibility to modelers, who can define the language through conditions, the main goal of a transformation and the desired side effects in an independent manner. This removes the need to consider complex interplays between rules and constraints, as in approaches which derive amalgamated rules which have to achieve a global effect with a single specification.

We notice that most transformations involve redirection of associations from one element to another, or changing the context for an element. The construction presented in the paper can be adapted to define *accumulators* and *distributors* of associations, which would collect all edges to be redirected, while deleting or constructing elements. Hence, such redirections might be taken as primitive constructs. The approach has been presented only for typed graphs. Extensions to graphs with inheritance and with attributes have to be explored, in particular for the case where identifiers are used to describe the associations of an element with others.

Future work for a full implementation of the approach will have to devise ways to reduce the number of rules, and particularly of NACs, generated. Hence optimizations pruning rules or NACs without effect in identifiable contexts should be studied.

Bibliography

[BGL08] P. Bottoni, E. Guerra, J. de Lara. Enforced generative patterns for the specification of the syntax and semantics of visual languages. *JVLC* 19(4):429–455, 2008.

[BQV06] P. Bottoni, P. Quattrocchi, D. Ventriglia. Constraining Concrete Syntax via Metamodel Information. In *Proc. IEEE VL/HCC 2006*. Pp. 85–88. IEEE CS Press, 2006.

[EEHP06] H. Ehrig, K. Ehrig, A. Habel, K.-H. Pennemann. Theory of Constraints and Application Conditions: From Graphs to High-Level Structures. *Fundam. Inform.* 74(1):135–166, 2006.

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

[EKTW06] K. Ehrig, J. M. Küster, G. Taentzer, J. Winkelmann. Generating Instance Models from Meta Models. In *Proc. FMOODS 2006*. LNCS 4037, pp. 156–170. Springer, 2006.

[GMT99] M. Goedicke, T. Meyer, G. Taentzer. ViewPoint-oriented software development by distributed graph transformation: towards a basis for living with inconsistencies. In *Proc. IEEE Requirements Engineering, 1999*. Pp. 92–99. 1999.

[HHT96] A. Habel, R. Heckel, G. Taentzer. Graph Grammars with Negative Application Conditions. *Fundam. Inform.* 26(3/4):287–313, 1996.

[HHT02] J. H. Hausmann, R. Heckel, G. Taentzer. Detection of conflicting functional requirements in a use case-driven approach: a static analysis technique based on graph transformation. In *Proc. ICSE '02*. Pp. 105–115. ACM Press, 2002.

[HMT+01] J. Howse, F. Molina, J. Taylor, S. Kent, J. Gil. Spider Diagrams: A Diagrammatic Reasoning System. *JVLC* 12(3):299–324, 2001.

[HP09] A. Habel, K.-H. Pennemann. Correctness of high-level transformation systems relative to nested conditions. *Math. Struc. in Comp. Sc.* 19(2):245–296, 2009.

[KKS97] H.-J. Kreowski, S. Kuske, A. Schürr. Nested graph transformation units. *Int. J. on SEKE* 7(4):479–502, 1997.

[MSW00] M. Münch, A. Schürr, A. J. Winter. Integrity Constraints in the Multi-paradigm Language PROGRES. In *Selected Papers from TAGT'98*. LNCS 1764, pp. 338–351. Springer, 2000.

[OEP08] F. Orejas, H. Ehrig, U. Prange. A Logic of Graph Constraints. In *Proc. FASE 2008*. LNCS 4961, pp. 179–198. Springer, 2008.

[Pen09] K.-H. Pennemann. *Development of Correct Graph Transformation Systems*. Ph.d thesis, Carl von Ossietzky Universität - Oldenburg, 2009.

[Ren04] A. Rensink. Representing First-Order Logic Using Graphs. In *Proc. ICGT*. LNCS 3256, pp. 319–335. Springer, 2004.

[RK09] A. Rensink, J.-H. Kuperus. Repotting the Geraniums: On Nested Graph Transformation Rules. *ECEASST - Proc. GT-VMT 2009* 18, 2009.

[Rus97] F. Ruskey. A Survey of Venn Diagrams. *Electronic Journal of Combinatorics*, 1997. www.combinatorics.org/Surveys/ds5/VennEJC.html.

Electronic Communications of the EASST
Volume of the Pre-proceedings of GT-VMT (2010)

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

Stochastic Graph Transformation with Regions

Paolo Torrini, Reiko Heckel and István Ráth

14 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

Stochastic Graph Transformation with Regions

Paolo Torrini[1], Reiko Heckel[2] and István Ráth[3]

[1] pt95@mcs.le.ac.uk
[2] reiko@mcs.le.ac.uk
University of Leicester

[3] rath@mit.bme.hu
Budapest University of Technology and Economics

Abstract: Graph transformation can be used to implement stochastic simulation of dynamic systems based on semi-Markov processes, extending the standard approach based on Markov chains. The result is a discrete event system, where states are graphs, and events are rule matches associated to general distributions, rather than just exponential ones. We present an extension of this model, by introducing a hierarchical notion of event location, allowing for stochastic dependence of higher-level events on lower-level ones.

Keywords: Graph Transformation, Stochastic Simulation, Topology

1 Introduction

Graph transformation combines the idea of graphs as a universal modelling paradigm with a rule-based approach to specify the evolution of systems [KK96]. Behaviour can be modelled in terms of labelled transition systems, where states are graphs and rule applications represent transitions. A discrete event system can be generally obtained by interpreting rule matches as events. Hierarchical graphs can be used to keep into account the spatial structure of graphs in terms of topological grouping, with advantages that have been underlined from the point of view of modelling and verification [BL09].

Stochastic graph transformation is applicable to probabilistic analysis and stochastic validation of graph-based modelling. Stochastic simulation can be particularly useful as validation technique when systems are too complex to be model-checked. It can be implemented relying on a discrete event system approach [CL08]. Transitions are labelled by scheduling times, randomly chosen according to given probability distributions — thus replacing stochastic determinism for indeterminism in the modelling.

A stochastic graph transformation system can be obtained, in a fairly simple form [HLM06], by associating each rule name with an exponential distribution. The associated Markov-chain analysis has been applied to integrated modelling of architectural reconfiguration and non- functional aspects of network models [Hec05]. However, this approach has some limits. Exponential distributions can express well the relative speed of processes, but are less than suited to describe phenomena that are characterised in terms of mean and deviation. Generalised stochastic graph transformation can answer this problem, allowing for general distributions to be associated with rule names [KL07] and more generally with rule matches [HT09, KTH09]. In the latter case,

assignment of probability distributions to events may depend e.g. on attributes of match elements. Generalised semi-Markov processes provide a discrete event-based semantic model for such systems.

In the physical world, events can often be described at different levels of spatial and causal granularity. Stochastic simulation based on graph transformation rests completely on the underlying graph transformation system in order to express causal dependencies, and deals with events as stochastically independent from each other. This fact can make it hard to take into account different levels of description, and to simplify the modelling of behaviour that involves very large numbers of transitions — although of course the use of global variables and derived attributes can ease such problems [HT09]. In this paper we consider a notion of structured stochastic simulation based on hierarchical graphs, making it possible to associate locations to events and to express a notion of granularity, thus allowing for stochastic dependence of higher-level events on co-located ones.

Regardless of the specific approach, a major stumbling block in implementing stochastic simulation based on graph transformation is the need to compute all the matches at each step. This is hard in principle — the subgraph homomorphism problem is known to be NP-complete, though feasible in many cases of interest. However, the cost of recomputing can be prohibitive — for this reason, we rely on incremental pattern matching based on a RETE-style algorithm as implemented in VIATRA [BHRV08] (a model transformation plugin of Eclipse). In [THR] we presented GraSS, a tool that extends VIATRA with a stochastic simulation control based on the SSJ Java libraries [LMV02]. By using a decoupled notion of graph hierarchy, it should be possible to implement hierarchical stochastic simulation in VIATRA/GraSS.

1.1 Hierarchical extensions

Hierarchy can be used in graph models in order to introduce a notion of topological grouping on model elements. Grouping information can be represented as a hierarchy graph, as distinct from the underlying one, relying on a decoupled approach [BKK05]. In the case of bigraphs, the modelling approach is based on a pairing of *place* graphs and *link* graphs, together with a complex notion of matching [Mil08].

Here we use topology to localise events, rather than elements, relying on a generic notion of rule matching. Formally, the hierarchy is a partial order ($\leq$). Topological grouping arises from rule matching in the underlying graph and the hierarchy. More precisely, a model consists of an underlying graph coupled with a place graph, where the latter is a directed acyclic graph (*dag*) from which the hierarchy arises, in which nodes are *places* and edges represent containment between hierarchy nodes (*hierarchical containment*). The two graphs are linked together by containment edges (*coupling containment*) that map underlying graph nodes to places. Regions are defined as downward-closed sets with respect to hierarchy, i.e. closed sets in the corresponding order topology [TSB02]. By mapping rule matches to regions, we can also facilitate dealing with sharing and independence for matches.

From the stochastic point of view, we want to use hierarchy to let lower-level events affect the assignment of probability distributions to higher-level ones. The assignment of probability distributions to events may then depend on information beyond that contained in the corresponding rule matches, and the probability of the application of a rule can depend on a larger subgraph

than its applicability condition. In this way, it should be possible to refine simulation without resorting to more complex models and without making reachability analysis harder.

We want to allow for dependency of the distribution assigned to an event (1) on the enabling of other events, and more generally on the number of enabled matches of a certain type — here called a *density* measure; (2) on the the scheduling of other events — called an *activity* measure.

A density measure boils down to counting matches, therefore it could be handled in a flat way by introducing more attributes — but this would mean making the model more complex. An activity measure can be trickier, as in principle it might lead to circular dependencies. This problem can be solved by requiring that stochastic dependency as well as the scheduling process follow the hierarchical order. Therefore, events cannot depend on higher level ones, and at each simulation step, the scheduling of lower-level events is computed before that of higher-level ones.

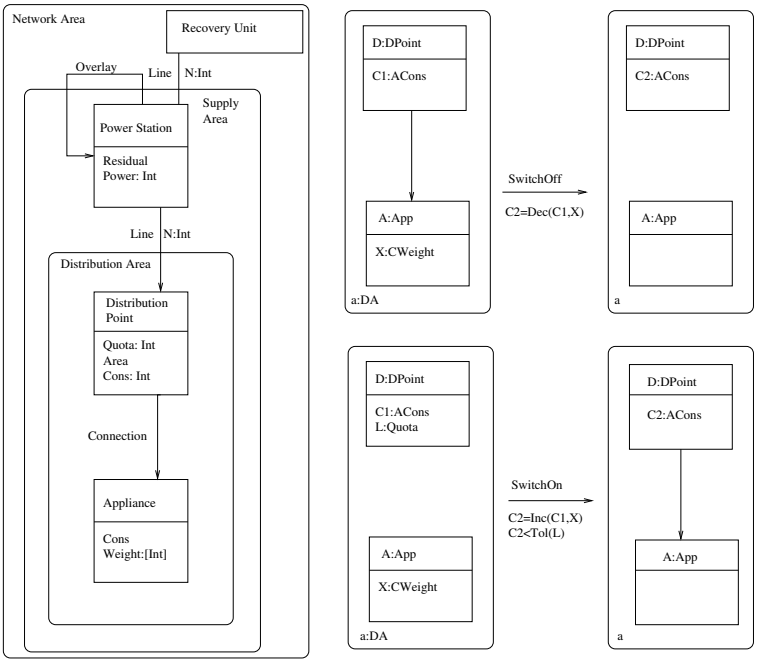
2 Example

We give a model example in which higher-level events may depend stochastically on large numbers of lower-level ones, by modelling a power grid. We consider a system in which each power source serves a number of distribution points, by allocating power quotas in a reconfigurable way. Appliances can be added to and removed from a distribution area, and they can be connected to and disconnected from distribution points, determining the level of consumption, which must remain within a tolerance of the quota. A power failure may occur when the quota is overstepped. A failure determines the disruption of the distribution point, with consequent loss of data, and it forces the intervention of a recovery unit. Actual reconfiguration is carried out following optimisation criteria that can be reflected stochastically in the application of the rule.

The model is based on the SPO approach, and uses typed graphs with attributes. A power station is connected to each of the distribution points by power lines denoted by multi-edges, i.e. sets of parallel edges represented as a single edge with an integer value. A station can reconfigure the capacity of each power line depending on the available power and the distribution area consumption — this takes place by changing the number of line edges, also updating residual power and local quota.

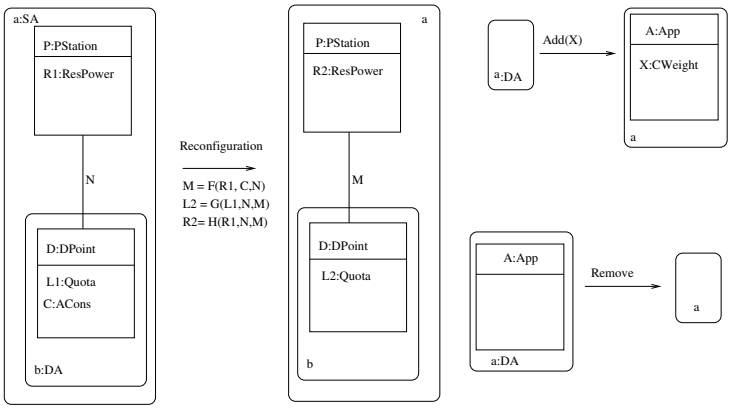
The spatial structure of the model is quite simple — there are three types of places: the network area, a supply area for each station, and a distribution area for each service point. Each place is represented as a rounded box. The hierarchy order $\leq$ is represented as containment (larger boxes are places higher up in the order, therefore associated with higher-level elements). The coupling order is also represented as containment in an obvious way — each underlying graph element (a square box) being coupled with the smallest place box it is contained in. In this example, the place graph is a tree. However, the notation can be easily extended to the *dag* case, by associating places to intersections. The symbols $Dec, Inc, Tol, Add, F, G, H, P$ in the pictures stand for defined functions.

The distinction between higher-level and lower-level events here is comparatively straightforward. The former ones are those associated with reconfigurations, failures and recoveries, and located in regions generated by supply areas. The latter ones are associated with adding, removing, switching on and off appliances, and are located in regions generated by distribution



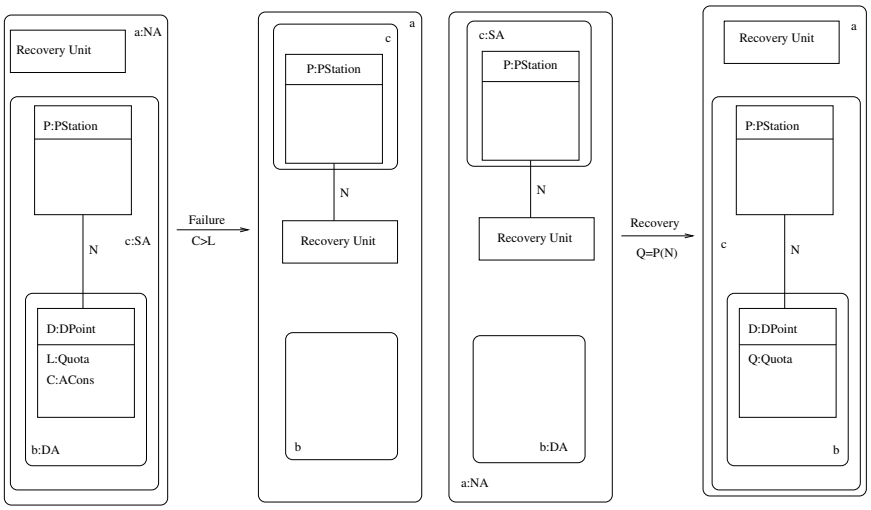
(a) Type graph, Switch-on/off

Figure 1: Type graph, Switch-on/off



(a) Reconfiguration, Add, Remove

Figure 2: Reconfiguration, Add, Remove



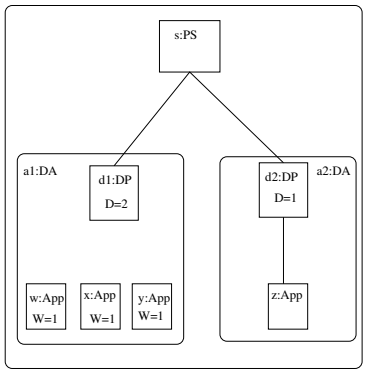
(a) Failure, Recovery

Figure 3: Failure, Recovery

areas. From the stochastic point of view, actions that depend heavily on aspects which are not modelled, such adding appliances, switching on and failure, may be associated with exponential distributions. Actions plausibly associated with variance values to be taken into consideration within the model, such as switching off and recovery, may be more naturally associated with normal distributions. Reconfiguration, on the other hand, may make expressing probabilistic dependence on the context quite attractive.

In the example, there are two possible matches for the reconfiguration rule — one with $a1$ and the other with $a2$ as distribution areas. In order to model stochastically a "smart" reconfiguration strategy, one could make the probability of application inversely dependant on the difference between quota and area consumption (a derived attribute associated with distribution points and denoted by D in the picture). However, if that is to be the only criteria, here there is little chance of modelling a high quality of service without changing the model. Given a high rate of switching on against a low one of appliance addition, the area $a1$ is more at risk than $a2$, in spite of the higher D value. This risk is essentially associated with the number of matches for *switch_on* in $a1$, and further than that — with their scheduling.

Of course it would be possible to retain information about the number of appliances in an area explicitly, by adding an attribute — however, apart from the need to extend the model, this way of capturing the density measure would not be quite natural in this case. The information could be retained by the service point — but when a failure occurs and the service is disrupted, the information would be lost. Moreover, it is difficult to think of a similar way to capture an activity measure. On the other hand, the knowledge embedded in the reconfiguration strategy might be actually based on estimates rather than precise data. Therefore, modelling it in terms of implicit stochastic dependence seems reasonable, and may be more realistic than representing it explicitly by means of attributes.



(a) Example

Figure 4: Example

3 Stochastic Graph Transformation

Stochastic graph transformation for semi-Markov process modelling requires special attention to the notion of match and persistence of matches. Although the running example is based on SPO, we prefer to give a general definition of typed graph transformation with respect to a generic approach, allowing for node type inheritance and negative application conditions. We then define a notion of structured GTS, by endowing graphs with hierarchy and hence derived topological structure.

3.1 Graph transformation

In existing axiomatic descriptions of graph transformation [KK96], a graph transformation approach is given by a class of graphs $\mathscr{G}$, a class of rules $\mathscr{R}$, and a family of binary relations $\Rightarrow_r \subseteq \mathscr{G} \times \mathscr{G}$ representing transformations by rules $r \in \mathscr{R}$. This general notion can be refined by introducing definitions of rule match and rule application, depending on a given approach (such as SPO and DPO).

We denote by M_r the set of *all* matches for a rule r, and by $M_R = \bigcup_{r \in R} M_r$ the union of all such sets of matches for all rules in $R \subseteq \mathscr{R}$ (with $M_{\mathscr{R}} = \bigcup_{r \in \mathscr{R}} M_r$). Similarly, $M_{R,G}$ is the subset of $M_{\mathscr{R}}$ made of all matches in a graph $G \in \mathscr{G}$.

We assume an $\mathscr{R} \times \mathscr{G}$-indexed family of sets of rule matches $M_{r,G}$ and extend $\Rightarrow_r$ to a partial function $\Rightarrow_{r,m} : \mathscr{G} \to \mathscr{G}$, such that $\Rightarrow_{r,m}(G)$ is defined if and only if $m \in M_{r,G}$. This captures the idea that rule application is well-defined and deterministic once a valid match m for r in G is found. In case r is equipped with application conditions, the match is deemed to satisfy them. As usual, we write $G \Rightarrow_{r,m} H$ for $\Rightarrow_{r,m}(G) = H$, and we say that this is the *transformation step* determined by the application of rule r to match m in graph G.

A graph transformation system (GTS) $\mathbf{G} = \langle R, G_0 \rangle$ consists of a set $R \subseteq \mathscr{R}$ of rules and an initial graph $G_0 \in \mathscr{G}$. A transformation in $\mathbf{G}$ is a sequence of rule applications $G_0 \Rightarrow_{r_1,m_1} G_1 \Rightarrow_{r_2,m_2} \cdots \Rightarrow_{r_n,m_n} G_n$ using rules in $\mathbf{G}$ with all graphs $G_i \in \mathscr{G}$. The set of the graphs reachable from G_0 with rules in R by a finite sequence of transformation steps is denoted by $\mathscr{L}_{\mathbf{G}}$. By $M_{\mathbf{G}}$ we denote

the set of all matches over all reachable graphs (indeed, in the following we always require this to be a set).

3.2 Persistence of rule matches

For algebraic approaches in which rule application is defined up to isomorphism, assumptions have to be made in order to make rule application deterministic, and in order to guarantee that matches form a proper set, unlike in more abstract presentations [EEPT06]. In order to capture the idea of a *concrete and deterministic* implementation of graph transformation, we assume that a choice of direct transformations is made such that the result of applying a rule to a match is unique.

Moreover, the notion of rule match needed for an implementation based on incremental pattern matching should be persistent through transformation, i.e. each rule match should have an identity that transcends individual states, satisfying the following property (*persistence property*): given the scheduling of two independent rule applications at matches m and m', once only one of them is applied, the other match remains itself enabled. In [KTH09, KL07] persistence was obtained by defining an equivalence relation on matches, relying on a conservative naming policy, according to which names are chosen consistently in consecutive transformations so to preserve the identities of nodes and edges where possible and never to reuse names from the past.

A similar result can be obtained here with a looser naming policy. Each graph G can be associated to the set $\bar{G}$ of its elements (nodes and edges). We can assume the typing includes connectivity information, by using dependent types for edges, as in [TH09] — then G can be represented as $\bar{G}$, and we may drop the distinction between them altogether. We do not fix a general approach. We just assume that rules preserve graph types. Then we can say that in general, a transformation step $t = G \Rightarrow_{r,m} H$ deletes a set of elements $D_t \subseteq G$ and creates a set of new elements $C_t \subseteq H$. From the deterministic character of rule application and the denumerability of the domain, $H \backslash C_t = \sigma_t(G \backslash D_t)$ follows, where σ_t is *the* (unique) renaming induced by t (assuming name spaces are disjoint).

The matches preserved by t can be defined as the largest set $M_{|t} \subseteq \{n \in M_{R,G} | n \cap D_t = \emptyset\}$ such that, under the assumptions made, $\sigma_t(M_{|t}) = \{n \in M_{R,H} | n \cap C_t = \emptyset\}$ or, equivalently, $M_{|t} = \{n | \sigma_t(n) \in M_{R,H} \wedge \sigma_t(n) \cap C_t = \emptyset\}$. We must define $M_{|t}$ as largest set rather than as equal, in order to allow for negative application conditions. We can abstract from renaming, by defining, for $n_1 \in M_{R,G'}, n_2 \in M_{R,G''}$, the symmetric relation $n_1 =_a n_2$ that holds whenever for all transformation steps t, if $t = G' \Rightarrow_{r,m} G''$ and $n_1 \in M_{|t}$, then $n_2 = \sigma_t(n_1)$, and if $t = G'' \Rightarrow_{r,m} G'$ and $n_2 \in M_{|t}$ then $n_1 = \sigma_t(n_2)$. We can now define the transitive closure using the least fixpoint operator (μ)

$$n_1 \equiv n_2 =_{df} \mu E.(n_1 = n_2) \vee (\exists n_3.E(n_1, n_3) \wedge n_3 =_a n_2)$$

It is a matter of routine showing that $\equiv$ is indeed an equivalence relation. Persistent matches over the set of the reachable graphs in $\mathbf{G}$ can now be defined as the quotient class $\mathscr{M}_{\mathbf{G}} = M_{\mathbf{G}} / \equiv$ — as one can see they satisfy the persistence property.

We can now introduce a notion of event. By $\mathscr{E}_{\mathbf{G}} = \{\langle r, m \rangle \mid m \in \mathscr{M}_{\mathbf{G}}, r \in R\}$ we denote the (enumerable) set of events $e_1, \ldots$, given by pairs of rules and match classes. We can write $\mathscr{E}_{r,G} = \{\langle r, m \rangle \mid m = [n] \wedge n \in M_G\}$ for the event matched by rule r in graph G and $\mathscr{E}_G = \bigcup_{r \in R} \mathscr{E}_{r,G}$ for all events in G. Clearly, an event and a graph determine a transformation step, and therefore

we can define $\Rightarrow_{\langle r,m\rangle}: \mathscr{G} \to \mathscr{G}$ as disjoint union of all $\Rightarrow_{r,n}$ for $n \in [m]$. We can also introduce a notion of *post-match* of an event $e = \langle r,m\rangle$ for each graph G s.t. $e \in \mathscr{E}_G$, with $post_G(e) = \langle m, RH_{e,G}\rangle$, where $RH_{e,G}$ is a subset of the underlying set of $\Rightarrow_e (G)$ that represents the post-condition of the transformation step — i.e. given $n \in [m]$ and $t = (\Rightarrow_{r,n} (G))$, we can define $RH_{e,G} = C_t \cup \sigma_t(n \setminus D_t)$.

3.3 Hierarchical structure

We say that a *state graph* G is *hierarchical* (in the decoupled sense [BKK05]) whenever the following two conditions are satisfied. (1) The graph G includes a distinguished acyclic directed subgraph S, called the *place graph* of G. The nodes of S are the *places*. The edges connecting them express hierarchical containment. (2) Each node of G has a single *location* edge (denoted by *loc*, expressing coupling containment) associating it to a place. We assume $loc(v) = v$ for all $v \in S$. We say that a place is empty when it is location only to itself.

We denote by $\leq$ the transitive closure of hierarchical containment. We will call *hierarchy* the resulting partial order $S_< = \langle S, \leq\rangle$. In practice, S is going to be mostly finite, however in theory it is sufficient to assume that $<$ is downward well-founded, i.e. that there are no countable infinite descending chains. We also assume $\leq$ to have finite degree (i.e. to be finitely branching). If in addition $\leq$ does not contain infinite ascending chains and is connected, finiteness follows, by Koenig's lemma. Localisation can be extended to edges of the underlying graph as follows: given an edge a between nodes v_1, v_2, we assume that the $\leq$-join $p = loc(v_1) \vee loc(v_2)$ exists, and that $loc(a) = p$.

We assume places have persistent identity in a strong sense (i.e. they cannot be renamed), and can be created and deleted only by means of dedicated rules (*place set rules*), that do not affect the underlying graph — therefore, only empty places can be created/deleted, whereas underlying transformation rules (transformation rules altogether) do not affect the place set. On the other hand, transformation rules may change the coupling as well as the hierarchy — hence affecting the localisation of underlying elements.

Given a set $K \subseteq S$ of places, we say that the *abstract region* generated by K is the set $reg(K) = \{x \in S | \exists y \in K \wedge x \leq y\}$, i.e. the downward-closed subset of S generated by K, and that the *concrete region* of K is the set $creg(K) = \{x \in G | loc(x) \in reg(K)\}$. Given an event $e = \langle r, [m]\rangle$ and a graph G s.t. $e \in \mathscr{E}_G$, we denote with $loc_G(e)$ (by overloading) the *location* of e, defined as the set of locations associated with the set of elements (*event elements*) that either are in the match or are to be deleted by the application of the rule, i.e. $loc_G(e) = \{loc(x) | x \in (m \cup D_t) \cap \bar{G}\}$ with $t = (\Rightarrow_e (G))$. We then say that $reg_G(e) = reg(loc(e))$ is the abstract region of e, and $graph_G(e) = creg(loc(e))$ is its concrete region.

Since we assumed that underlying transformation rules do not create/delete places, it follows that not only the match, but also the post-match of the rule falls within the region generated by the location of the corresponding event — however, the region associated to the post-match may be topologically different from the initial one, though generated by the same places. The definition of event location can be simplified in the case of DPO, since there the elements to be deleted are always part of the match. On the other hand, one of the reasons to have regions is in fact to provide a notion of modularity weaker than the DPO one — essentially, by giving a way to identify the subgraph that has been affected by the transformation (i.e. the concrete region),

and that by definition includes the nodes that may have lost some edge, the general idea being to wrap the match of an SPO rule into an abstract region — e.g. the *Failure* rule in the running example — thus making it comparatively modular, if only in a weak sense.

The abstract regions on S can be associated with the closed sets of the order topology $\mathscr{O} = \langle S, \leq \rangle$ determined by $\leq$ on S (*event space*), with global region $\top$ and empty region $\bot$. Relying on a standard construction, dual to that based on open sets [Vic89, TSB02], the closed sets are indeed the subsets of S that are downward-closed with respect to $\leq$. Inclusion $\subseteq$ then determines a partial order $\sqsubseteq$ on the regions (*granularity*). From the downward well-foundedness of $<$, it follows that $\sqsubset$ is similarly well-founded, i.e. it has no countable infinite descending chains. We may call *atomic* regions the smallest ones with respect to $\sqsubseteq$.

We say that an event e is *lower-level* with respect to an event e' (resp. e' is *higher-level* wrt e) iff $e \sqsubset e'$, we say that e is *located* in e' iff $e \sqsubseteq e'$. Clearly, $e \sqsubseteq e'$ iff $graph(e) \subseteq graph(e')$ — i.e. the event order is reflected in the underlying graph. Different abstraction techniques based on partitioning graphs and unfolding event structures have been extensively investigated [BKR05, BBER08, BCK08].

3.4 Stochastic Modelling

In the following we use types that depend on terms [Bar92], assuming GTS elements can be treated as typed terms, in order to keep formal expressions concise, and we write $\Pi x : \alpha.\beta$ rather than $\alpha \to \beta$ when β depends on x. We also implicitly assume that events are typed by the associated rules. A generalised stochastic graph transformation system (GSGTS) can be defined [HT09] as structure $\mathbf{S} = \langle \mathbf{G}, \Delta \rangle$ where $\mathbf{G}$ is a GTS, $\Delta : \Pi e \in \mathscr{E}_{\mathbf{G}}.Dist(e)$ is a distribution assignment, which associates with every event a cumulative distribution function (*cdf*, a function from real numbers to probability values), and we denote by $Dist_G(e) \subseteq Reals \to [0,1]$ the type of $cdf_G(e)$ as the *cdf* assigned to event e in G.

The behaviour of a GSGTS can be described as a stochastic process over continuous time, where reachable graphs form the state space and the application of transformation rules defines state transitions as instantaneous events. More precisely, a rule enabled by a match defines an event associated with an independent random variable (*timer*, or scheduling time) which represents the time expected to elapse (*scheduled delay*) before the rule is applied to the match. As the simulation is executed, the timer is randomly set according to the static specification provided by the *cdf* of the corresponding event — in the implementation, this involves a call to a pseudo-random number generator (RNG).

We want to extend the definition of GSGTS in order to keep into account the stochastic dependency of higher-level events on lower-level ones — e.g. of *Reconfiguration* matches on *Add* and *SwitchOn* matches in the running example. This involves introducing a dependency of the distribution assignment on the state. Therefore, we assume the *cdf* associated with an event can depend (1) on the number of events that are located in it, i.e. the value of $cdf_G(e)$ may depend on $\{e' | e' \sqsubseteq e\}$ (*local density*), and (2) on the values set for the timers of lower-level events, i.e. the value of $cdf_G(e)$ may depend on $\{timer(e') | e' \sqsubset e\}$ (*local activity*). From the point of view of the implementation, this means that, in each state G, the scheduling (i.e. the setting of timers) has to respect $\sqsubset_G$ (i.e. the granularity $\sqsubset$ in G). We then define a type

$$LDens_G(e) = \Pi r.Num(r)$$

with $Num(r) \subseteq Nat$, for the cardinalities of the rule match sets (one for each rule) on which the event e may depend on, and we assume to have a match counting function $count_G(e) : LDens_G(e)$. We also assume to have a scheduling function $sched_G : \Pi e.Dist_G(e) \to Time(e)$, with $Time(e) \subseteq Reals$, which assigns a value to each timer, given its *cdf*. A partial scheduling will be a function of type

$$LAct_G =_{df} \Pi e_1 e_2.e_2 \sqsubset_G e_1 \to Time(e_2)$$

The distribution assignment δ may depend on partial scheduling (activity) and match counting (density). Therefore the type of the abstract distribution assignment can be

$$\delta : \Pi G.\Pi e.(LDens_G(e) \times LAct_G(e)) \to Dist_G(e)$$

Essentially, δ must be provided in order to complete the model. Partial scheduling $\pi_G : \Pi e.LDens_G(e) \times LAct_G(e)$ can now be defined by recursion (assuming δ is model-specific)

$$\pi_G =_{df} \mu f.\lambda e_1.(count_G(e_1),\ \lambda e_2.\ \text{if } e_2 \sqsubset_G e_1 \text{ then } (sched_G(e_2)(\delta(G)(e_2)(f(e_2)))))$$

Finally, the distribution assignment $\Delta : \Pi e.\Pi G.Dist_G(e)$ can be defined as

$$\Delta =_{df} \lambda e.\lambda G.\delta(G)(e)(\pi_G(e))$$

This shows that Δ can be defined recursively, given $sched, count, \delta$, relying on the well-foundedness of $\sqsubset$. Then $\mathbf{H} = \langle \mathbf{G}, \delta \rangle$ is a *stochastic hierarchical graph transformation system* (HSGTS) when $\mathbf{G}$ is a GTS and δ is a function specified as above, so that the function that assigns a continuous *cdf* to each event for each reachable graph can be derived as $\Delta : \Pi e \in \mathscr{E}_{\mathbf{G}}, \Pi G \in \mathscr{L}_{\mathbf{G}}.Dist_G(e)$. Note that dependence on graphs can be refined into dependence on concrete regions, by replacing the definition of Δ with one of $\Delta' : \Pi e \in \mathscr{E}_{\mathbf{G}}, \Pi G \in \mathscr{L}_{\mathbf{G}}.Dist_{graph_G(e)}(e)$.

We also find it useful to allow for rule constructors, i.e. $c : X \to R$ with X finite domain, as syntactic sugar for a set of rules $r_1 = c(x_1), \ldots, r_k = c(x_k)$, where k is the cardinality of X — e.g. *Add(X)* in the example.

4 Stochastic Simulation

We can define an operational interpretation of HSGTS in terms of semi-Markov processes, following an approach already used in [THR, KTH09, KL07]. We rely on a representation of stochastic processes as discrete event systems [CL08], and define a translation of HSGTS into them.

Semi-Markov processes are a generalisation of Markov processes — indeed, they can be regarded as Markov chains with timers. Markov processes enjoy the memoryless property — each event depends on the current state only, it is independent from past states as well as from the time spent in the current one. When a Markov process is regarded as discrete event system, the timers

associated with events, and consequently also interevent times, must be exponentially distributed — therefore they can be restarted at each change of state without any loss.

In semi-Markov processes, timers (hence also interevent times) can be generally distributed. This means that events are independent of past states, but may depend on timers that have been set in previous states. More formally, a semi-Markov process can be defined as a process generated by a *generalised semi-Markov scheme* (GSMS) [DK05]. The relationship between semi-Markov processes and GSMS is altogether similar to that between Markov processes and Markov chains — indeed, a process generated by a GSMS where all timers are exponentially distributed variables is stochastically equivalent to a continuous-time Markov chain.

We need to define a structure that is syntactically more general than a GSMS, insofar as we want to allow for distribution assignments to depend on states. A *hierarchical semi-Markov scheme* (HSMS) is here a structure

$$\mathscr{P} = \langle\, Z,\, E,\, enabled : Z \to \wp(E),\, new : Z \times E \to Z,\, cdfAsg : E \to Z \to Reals \to [0,1], \\ \sqsubseteq^S : Z \to (E \times E) \to bool,\, s_0 : Z \,\rangle$$

where Z is a set of system states; E is a set of timed events; *enabled* is the activation function, so that $enabled(s)$ is the finite set of active events associated with state s; *new* is a partial function depending on states and events that represent transitions; *cdfAsg* is the distribution assignment, so that $cdfAsg(e)(s)$ gives a *cdf* of the scheduled delay of e at state s; $\sqsubseteq^S(s)$ is a well-founded order (*schedule-making order*) on the enabled events; and s_0 is the initial state.

Given a HSGTS $\mathbf{H} = \langle R, G_0, \delta \rangle$, we can define its translation to an HSMS $\mathscr{P}_{\mathbf{H}}$ as follows

- $Z = \mathscr{L}(\langle R, G_0 \rangle)$ is the set of graphs reachable from G_0 by rules in R;
- $E = \mathscr{E}_{\mathbf{G}}$ is the set of possible events for $\mathbf{G} = \langle R, G_0 \rangle$;
- $enabled(G) = \{e \mid e \in \mathscr{E}_G\}$ is the set of all events enabled in graph G;
- the transition function is defined by $new(G,e) = H$ iff $G \Rightarrow_e H$;
- distribution $cdfAsg(e)(G)$ is given by $\Delta(e)(G)$;
- the scheduling order $\sqsubseteq^S(G)$ is defined as event order $\sqsubseteq_G$
- s_0 is G_0

This embedding can be used as static framework for the definition of a simulation algorithm that is adequate with respect to system runs, in the sense that there is a one-to-one correspondence between the runs of the original HSGTS and those of the resulting HSMS, and therefore correct and complete with respect to reachability. The algorithm, based on the general scheduling scheme given in [CL08], can be described as follows.

- Initial step
  1. The simulation time is initialised at 0.

2. The set of the enabled events $A = enabled(s_0)$ is computed.
3. The schedule-making order $\sqsubseteq^S(s_0)$ is computed and checked for well-foundedness.
4. For each event $e \in A$, a scheduling time t_e is computed by an RNG as random delay value d_e depending on the probability distribution function $cdfAsg(e)(s_0)$;
5. The enabled events with their scheduling times are collected in the scheduled event list $l_{s_0} = \{(e,t_e) | e \in A\}$, ordered by time values.

- For each successive step — given the current state $s \in Z$ and the associated scheduled event list $l_s = \{(e,t) | e \in active(s)\}$
  1. the first element $k = (e,t)$ is removed from l_s;
  2. the simulation time t_S is updated by increasing it to t;
  3. the new state s' is computed as $s' = new(s,e)$;
  4. the list $m_{s'}$ of the surviving events is computed, by removing from l_s all the elements become disabled, i.e. all the elements (z,x) of l_s such that $z \notin enabled(s')$;
  5. The schedule-making order $\sqsubseteq^S(s')$ is computed and checked for well-foundedness.
  6. a list $n_{s'}$ of the newly enabled events is built, containing a single element (z,t_z) for each event z such that $z \in enabled(s') \backslash enabled(s)$ and has scheduling time $t_z = t_S + d_z$, where d_z is a random delay value given by an RNG depending on the distribution function $cdfAsg(s')(z)$;
  7. the new scheduled event list $l_{s'}$ is obtained by reordering the concatenation of $m_{s'}$ and $n_{s'}$ with respect to time values

Part of the complexity of the algorithm is hidden in the recursive definition of *cdfAsg*. This requires the schedule-making order $\sqsubset^S$ to be well-founded — which of course in our translation is guaranteed by the assumptions on the hierarchy $\leq$. Apart from that, the algorithm relies on scheduling, implemented by calls to an RNG, and on a standard match counting function.

5 Further work

Expressing stochastic dependencies associated with density and activity measures — as in the example — can be useful to model situations in which specific events depend on large numbers of co-located ones, as in describing biochemical processes at different levels of detail (e.g. molecular and cellular). Graph transformation can be good at tracking individual processes — however, there are aspects that can be modelled more efficiently in terms of mass effect and differential equations [Car08]. Therefore, the general capability of expressing presence of reactants and reactions in a region can be useful.

In the implementation, applying few high-level rules rather than many low-level ones could ease the costly business of updating incremental data-structures. As presently implemented in GraSS [THR], flat scheduling is carried out rather independently of the RETE network. Proceeding that way, hierarchical scheduling might turn out to be expensive. Reflection of hierarchy into the underlying graph could be handled quite more efficiently, however, by introducing a

further level of incrementality based on a *live transformation* approach [RBOV08], i.e. by continuously maintaining spatial information as part of the transformation context, so that changes to the spatial structure can be instantly mapped to the underlying graph.

6 Conclusion

Stochastic simulation is a promising field of application for graph transformation techniques. We have argued that structured graphs can be useful for stochastic simulation. We have focussed on probabilistic dependency of events on co-located ones, in order to make stochastic modelling more expressive, and to define models that are more flexible from the point of view of simulation, by offering more options without resorting to structural changes.

We have shown that hierarchy can be used to define a topological order on events, allowing for a weak notion of modularity and an increase in expressiveness with respect to stochastic dependencies. This extension can be embedded in discrete event models of stochastic processes. We think that an implementation of stochastic simulation along this lines could take great benefit from an appropriate use of references in incremental pattern-matching.

References

[Bar92] H. P. Barendregt. Lambda Calculi with Types. In Abramsky et al. (eds.), *Handbook of Logic in Computer Science*. Volume 2, pp. 117–309. Oxford – Clarendon Press, 1992.

[BBER08] J. Bauer, I. Boneva, K. M. E., A. Rensink. A modal-logic based graph abstraction. In *ICGT 2008*. 2008.

[BCK08] P. Baldan, A. Corradini, B. König. A framework for the verification of infinite-state graph transformation systems. *Information and Computation* 206:869–907, 2008.

[BHRV08] G. Bergmann, A. Horváth, I. Rath, D. Varró. A benchmark evaluation of incremental pattern matching in graph transformation. In *International Conference on Graph Transformation*. LNCS 5214, pp. 396–410. 2008.

[BKK05] G. Busatto, H.-J. Kreowski, S. Kuske. Abstract hierarchical graph transformation. *Mathematical Structures in Computer Science* 15(4):773–819, 2005.

[BKR05] P. Baldan, B. König, A. Rensink. Graph grammar verification through abstraction. In *Graph transforamtion and process algebras for modeling distributed and mobile systems*. Dagstuhl Seminar 04241. 2005.

[BL09] R. Bruni, A. Lluch Lafuente. Ten Virtues of Structured Graphs. In *GT-VMT'09*. 2009.

[Car08] L. Cardelli. Artificial Biochemistry. In Springer (ed.), *Algorithmic Bioprocesses*. LNCS, 2008.

[CL08] C. G. Cassandras, S. Lafortune. *Introduction to discrete event systems*. Kluwer, 2008.

[DK05] P. R. D'Argenio, J.-P. Katoen. A theory of stochastic systems part I: Stochastic automata. *Inf. Comput.* 203(1):1–38, 2005.

[EEPT06] H. Ehrig, K. Ehrig, U. Prange, G. Taentzer. *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series)*. Springer, 2006.

[Hec05] R. Heckel. Stochastic Analysis of Graph Transformation Systems: A Case Study in P2P Networks. In *Proc. Intl. Colloquium on Theoretical Aspects of Computing (ICTAC'05)*. LNCS 3722, pp. 53–69. Springer-Verlag, 2005.

[HLM06] R. Heckel, G. Lajios, S. Menge. Stochastic graph transformation systems. *Fundamenta Informaticae* 72:1–22, 2006.

[HT09] R. Heckel, P. Torrini. Stochastic Modelling and Simulation of Mobile Systems. 2009. submitted for publication.

[KK96] H.-J. Kreowski, S. Kuske. On the Interleaving Semantics of Transformation Units - A Step into GRACE. Pp. 89 – 106. 1996.

[KL07] P. Kosiuczenko, G. Lajios. Simulation of generalised semi-Markov processes based on graph transformation systems. *Electronic Notes in Theoretical Computer Science* 175:73–86, 2007.

[KTH09] A. Khan, P. Torrini, R. Heckel. Model-based Simulation of VoIP Network Reconfigurations using Graph Transformation Systems. In Corradini and Tuosto (eds.), *Intl. Conf. on Graph Transformation (ICGT) 2008 - Doctoral Symposium*. Electronic Communications of the EASST 16. 2009.

[LMV02] P. L. L'Ecuyer, L. Meliani, J. Vaucher. SSJ: a framework for stochastic simulation in Java. In *Proceedings of the 2002 Winter Simulation Conference*. Pp. 234–242. 2002.

[Mil08] R. Milner. Bigraphs and Their Algebra. *Electr. Notes Theor. Comput. Sci.* 209:5–19, 2008.

[RBOV08] I. Rath, G. Bergmann, A. Okrős, D. Varró. Live model transformations driven by incremental pattern matching. In *ICMT'08*. LNCS 5063, pp. 107–121. Springer, 2008.

[TH09] P. Torrini, R. Heckel. Towards an embedding of Graph Transformation in Intuitionistic Linear Logic. *CoRR* abs/0911.5525, 2009.

[THR] P. Torrini, R. Heckel, I. Ráth. Stochastic Simulation of Graph Transformation Systems. accepted at FASE'10.

[TSB02] P. Torrini, J. G. Stell, B. Bennett. Mereotopology in second-order and modal extensions of intuitionistic propositional logic. *Journal of Applied No-Classical Logic* 12(3–4):490–525, 2002.

[Vic89] S. Vickers. *Topology via logic*. Cambridge University Press, 1989.

Electronic Communications of the EASST
Volume of the Pre-proceedings of GT-VMT (2010)

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

Co-tabulations, Bicolimits and Van-Kampen Squares in Collagories

Wolfram Kahl

14 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

Co-tabulations, Bicolimits and Van-Kampen Squares in Collagories

Wolfram Kahl[1]

[1]kahl@cas.mcmaster.ca, http://sqrl.mcmaster.ca/~kahl/
Department of Computing and Software, McMaster University, Hamilton, Ontario, Canada

Abstract: We previously defined collagories essentially as "distributive allegories without zero morphisms". Collagories are sufficient for accommodating the relation-algebraic approach to graph transformation, and closely correspond to the adhesive categories important for the categorical DPO approach to graph transformation.

Heindel and Sobociński have recently characterised the Van-Kampen colimits used in adhesive categories as bicolimits in span categories.

In this paper, we study both bicolimits and lax colimits in collagories. We show that the relation-algebraic co-tabulation concept is equivalent to lax colimits of difunctional morphisms and to bipushouts, but much more concise and accessible. From this, we also obtain an interesting characterisation of Van-Kampen squares in collagories.

Keywords: Relation-algebraic graph transformation, Collagories, Allegories, Pushout, Adhesive categories

1 Introduction

One of the hallmarks of the relation-algebraic approach to graph transformation [Kaw90, Kah01, Kah04] is that it allows an abstract characterisation of the gluing condition for the double pushout approach. Nevertheless, the categorical approach to graph transformation has continued to use the node-and-edge-based formulation of the gluing condition even in the handbook chapter [CMR+97]. Recently, the literature of the categorical approach, starting essentially with [EPPH06] has adopted the "adhesive categories" of Lack and Sobociński [LS04], where however the details of the gluing condition are completely sidestepped.

In [Kah09a], we introduced *collagories* essentially as "distributive allegories without zero morphisms". We redeveloped in collagories the fundamentals of the relation-algebraic approach to graph transformation, and showed that adhesive categories arise, and also that bitabular collagories share the most important construction principles, such as slice and co-slice category constructions, with adhesive categories.

Inspired by Heindel and Sobociński's characterisation of van Kampen squares as bicolimits in the bicategory of spans [HS09], we establish in this paper (Sect. 6) the connections between our co-tabulations and bicolimits in collagories, succeding to show that the co-tabulation characterisation of pushouts, which essentially goes back to Kawahara [Kaw90], has a *precise* categorical counterpart in bipushouts, and, even more closely, in lax colimits of difunctional morphisms in a collagory context.

We also succeed in providing, in Sect. 7, an original collagory-theoretic characterisation of van Kampen squares, significantly advancing over the results of [Kah09a, Kah09b].

2 Categories, Allegories

This section only serves to fix notation and terminology for standard concepts, see [FS90, SS93, Kah04]. Like Freyd and Scedrov and a slowly increasing number of categorists, we denote composition in "diagram order" not only in relation-algebraic contexts, where this is customary, but also in the context of categories. We will always use the infix operator ";" to make composition explicit: $R;S = \mathscr{A} \xrightarrow{R} \mathscr{B} \xrightarrow{S} \mathscr{C}$.

Definition 2.1. A *category* $\mathbf{C}$ is a tuple $(\mathsf{Obj}_\mathbf{C}, \mathsf{Mor}_\mathbf{C}, \mathsf{src}, \mathsf{trg}, \mathbb{I}, ;)$ where

- $\mathsf{Obj}_\mathbf{C}$ is a collection of *objects*.
- $\mathsf{Mor}_\mathbf{C}$ is a collection of *arrows* or *morphisms*.
- src (resp. trg) maps each morphism to its source (resp. target) object.
  Instead of $\mathsf{src}(f) = \mathscr{A} \wedge \mathsf{trg}(f) = \mathscr{B}$ we write $f : \mathscr{A} \to \mathscr{B}$.
  The collection of all morphisms f with $f : \mathscr{A} \to \mathscr{B}$ is denoted as $\mathsf{Mor}_\mathbf{C}[\mathscr{A}, \mathscr{B}]$ and also called a *homset*.
- ";" is the binary *composition* operator, and composition of two morphisms $f : \mathscr{A} \to \mathscr{B}$ and $g : \mathscr{B}' \to \mathscr{C}$ is defined iff $\mathscr{B} = \mathscr{B}'$, and then $(f;g) : \mathscr{A} \to \mathscr{C}$; composition is associative.
- $\mathbb{I}$ associates with every object $\mathscr{A}$ a morphism $\mathbb{I}_\mathscr{A}$ which is both a right and left unit for composition. □

Definition 2.2. An *ordered category* is a category $\mathbf{C}$ such that

- for each two objects $\mathscr{A}$ and $\mathscr{B}$, the relation $\sqsubseteq_{\mathscr{A},\mathscr{B}}$ is a partial order on $\mathsf{Mor}_\mathbf{C}[\mathscr{A}, \mathscr{B}]$ (the indices will usually be omitted), and
- composition is monotonic with respect to $\sqsubseteq$ in both arguments. □

For homsets that have least or greatest elements, we introduce corresponding notation:

Definition 2.3. In an ordered category, for each two objects $\mathscr{A}$ and $\mathscr{B}$ we introduce the following notions:

- If the homset $\mathsf{Mor}_\mathbf{C}[\mathscr{A}, \mathscr{B}]$ contains a greatest element, this is denoted $\top_{\mathscr{A},\mathscr{B}}$.
- If the homset $\mathsf{Mor}_\mathbf{C}[\mathscr{A}, \mathscr{B}]$ contains a least element, this is denoted $\bot_{\mathscr{A},\mathscr{B}}$. □

For these extremal morphisms and for identities we frequently omit indices where these can be induced from the context.

Definition 2.4. An *ordered category with converse*, or *OCC*, is an ordered category such that

- each morphism $R : \mathscr{A} \to \mathscr{B}$ has a *converse* $R^\smile : \mathscr{B} \to \mathscr{A}$,
- the *involution equations* hold for all $R : \mathscr{A} \to \mathscr{B}$ and $S : \mathscr{B} \to \mathscr{C}$:
$$(R^\smile)^\smile = R \qquad \mathbb{I}_\mathscr{A}^\smile = \mathbb{I}_\mathscr{A} \qquad (R;S)^\smile = S^\smile;R^\smile$$
- conversion is monotonic with respect to $\sqsubseteq$. □

Many standard properties of relations can be characterised in the context of OCCs [Kah04]:

Definition 2.5. A morphism $R : \mathscr{A} \to \mathscr{B}$ in an OCC is called:

- *univalent* iff $R^\smile;R \sqsubseteq \mathbb{I}_\mathscr{B}$,
- *total* iff $\mathbb{I}_\mathscr{A} \sqsubseteq R;R^\smile$,
- *injective* iff $R;R^\smile \sqsubseteq \mathbb{I}_\mathscr{A}$,
- *surjective* iff $\mathbb{I}_\mathscr{B} \sqsubseteq R^\smile;R$,
- a *mapping* iff it is univalent and total,

- *bijective* iff it is injective and surjective,
- *difunctional* iff $R;R^\smile;R \sqsubseteq R$. □

For an OCC $\mathbf{C}$, we write $\mathsf{Map}\,\mathbf{C}$ for the sub-category of $\mathbf{C}$ that contains only the mappings as arrows.

Difunctionality will play an important rôle in this paper; a concrete relation, understood as a Boolean matrix, is difunctional iff it can be rearranged into "loose block-diagonal form", with full rectangular blocks such that there is no overlap between different blocks in either direction. (See [SS93, 4.4] for more about difunctionality).

For endomorphisms, there are a few additional properties of interest:

Definition 2.6. A morphism $R : \mathscr{A} \to \mathscr{A}$ in an OCC is called:

- *reflexive* iff $\mathbb{I} \sqsubseteq R$,
- *transitive* iff $R;R \sqsubseteq R$, and *idempotent* iff $R;R = R$,
- *co-reflexive* or a *sub-identity* iff $R \sqsubseteq \mathbb{I}_{\mathscr{A}}$,
- *symmetric* iff $R^\smile \sqsubseteq R$,
- an *equivalence* iff it is symmetric, reflexive and transitive. □

Lemma 2.7. If $\mathscr{B} \xleftarrow{P} \mathscr{A} \xrightarrow{Q} \mathscr{C}$ is a span and $P^\smile;Q$ is difunctional, then $P;P^\smile;Q;Q^\smile$ is idempotent. If P and Q are moreover total, then $P;P^\smile;Q;Q^\smile$ is an equivalence.

PROOF: The first claim is immediate: $P;P^\smile;Q;Q^\smile;P;P^\smile;Q;Q^\smile = P;P^\smile;Q;Q^\smile$.

For the second claim, reflexivity is obvious from totality, and the first claim implies transitivity, and, together with totality, also symmetry:

$$Q;Q^\smile;P;P^\smile = \mathbb{I}_{\mathscr{A}};Q;Q^\smile;P;P^\smile;\mathbb{I}_{\mathscr{A}} \sqsubseteq P;P^\smile;Q;Q^\smile;P;P^\smile;Q;Q^\smile = P;P^\smile;Q;Q^\smile \qquad \square$$

While Freyd and Scedrov [FS90] derive the homset ordering in their allegories from the meet operation, we define allegories on top of ordered categories — the composition operator has higher precedence than all other binary operators.

Definition 2.8. An *allegory* is an OCC such that

- each homset is a lower semilattice with binary meet $\sqcap$.
- for all $Q : \mathscr{A} \to \mathscr{B}$, $R : \mathscr{B} \to \mathscr{C}$, and $S : \mathscr{A} \to \mathscr{C}$, the *modal rule* holds:

$$Q;R \sqcap S \sqsubseteq (Q \sqcap S;R^\smile);R \ . \qquad \square$$

The most well-known allegory is the category *Rel* of sets with relations and standard relational operations. Logical theories give rise to allegories of *derived predicates* [FS90, App. B]. A simpler case of that are the allegories arising from Σ-algebras (over some signature Σ) as objects, and with "relational Σ-homomorphisms", i.e. bisimulations in the sense of [Kah04], as morphisms.

In allegories, one can define domain and range operators:

Definition 2.9. For every morphism $R : \mathscr{A} \leftrightarrow \mathscr{B}$ in an allegory, we define $\mathsf{dom}\,R : \mathscr{A} \leftrightarrow \mathscr{A}$ and $\mathsf{ran}\,R : \mathscr{B} \leftrightarrow \mathscr{B}$ as:

$$\mathsf{dom}\,R := \mathbb{I}_{\mathscr{A}} \sqcap R;R^\smile \qquad\qquad \mathsf{ran}\,R := \mathbb{I}_{\mathscr{B}} \sqcap R^\smile;R \qquad \square$$

3 Collagories

$\kappa\acute{o}\lambda\lambda\alpha$: glue

In Freyd and Scedrov's treatment, although allegories are not required to have zero-ary meets, distributive allegories are required to have zero-ary joins (least elements) together with distributivity of composition over them, that is, the zero law $\bot ; R = \bot$. In [Kah09a], we introduced an intermediate concept that does not assume anything about zero-ary joins:

Definition 3.1. A *collagory* is an allegory where each homset is a distributive lattice with binary join $\sqcup$, and composition distributes over binary joins from both sides. □

We directly axiomatise difunctional closure, without introducing Kleene star:

Definition 3.2. A *difunctionally closed collagory* is a collagory where, there is an additional unary operation $\_^{\boxtimes}$ which satisfies the following axioms for all $R : \mathscr{A} \to \mathscr{B}$, $Q : \mathscr{C} \to \mathscr{A}$, and $S : \mathscr{B} \to \mathscr{C}$: $Q' : \mathscr{C} \to \mathscr{B}$, and $S' : \mathscr{A} \to \mathscr{C}$:

$$R^{\boxtimes} \;=\; R \sqcup R^{\boxtimes} ; (R^{\boxtimes})^{\smile} ; R^{\boxtimes} \qquad \text{recursive definition}$$

$$Q ; R \;\sqsubseteq\; Q' \;\wedge\; Q' ; R^{\smile} ; R \;\sqsubseteq\; Q' \;\Rightarrow\; Q ; R^{\boxtimes} \;\sqsubseteq\; Q' \qquad \text{right induction}$$

$$R ; S \;\sqsubseteq\; S' \;\wedge\; R ; R^{\smile} ; S' \;\sqsubseteq\; S' \;\Rightarrow\; R^{\boxtimes} ; S \;\sqsubseteq\; S' \qquad \text{left induction}$$

We further define $R^{\blacktriangleright} : \mathscr{A} \to \mathscr{A}$ and $R^{\blacktriangleleft} : \mathscr{B} \to \mathscr{B}$ as:

$$R^{\blacktriangleright} := \mathbb{I} \sqcup R^{\boxtimes} ; (R^{\boxtimes})^{\smile} \qquad \text{and} \qquad R^{\blacktriangleleft} := \mathbb{I} \sqcup (R^{\boxtimes})^{\smile} ; R^{\boxtimes} \;.$$ □

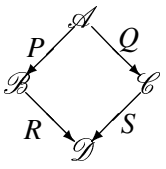
In a difunctionally closed collagory, the operation $\_^{\boxtimes}$ produces difunctional closures [Kah09a].

Requiring least morphisms satisfying zero laws turns collagories into distributive allegories, which still heave a much weaker theory than relations in a topos, so graph structures (unary algebras) with relational graph homomorphism in particular also form collagories.

In [Kah09a], we showed that the absence of the zero laws enables the presence of constant symbols (allowing for example pointed sets), and also that restrictions to sub-collagories in signature reducts (for example fixing label sets) and nested algebra constructions (interpreting signatures in the mapping categories of arbitrary collagories instead of just in $\mathsf{Map}\mathit{Rel}$) both construct new collagories. These constructions are directly useful for concrete modelling tasks, and for implementation of the resulting models as data structures; they also subsume the construction methods presented by Lack and Sobociński [LS04] for adhesive categories, in particular comprising clice and co-slice category construction.

4 Tabulations and Co-tabulations

Central to the connection between pullbacks and pushouts in categories of mappings on the one hand and constructions in relational theories on the other hand is the fact that a square of mappings commutes iff the "relation" induced by the source span is contained in that induced by the target co-span. The proof of this does not need the modal rule.



Lemma 4.1. [FS90, 2.146] Given a square of mappings in an allegory as drawn above, we have $P ; R = Q ; S$ iff $P^{\smile} ; Q \sqsubseteq R ; S^{\smile}$. □

This provides a first hint that in the relational setting, the identity of the two mappings P and Q does not matter when looking for a pushout of the span $\mathscr{B} \xleftarrow{P} \mathscr{A} \xrightarrow{Q} \mathscr{C}$ — we only need to consider the diagonal $P^\smile ; Q$. Dually, when looking for a pullback of the co-span $\mathscr{B} \xrightarrow{R} \mathscr{D} \xleftarrow{S} \mathscr{C}$, only $R ; S^\smile$ needs to be considered. The gap between the two ways of calculating the horizontal diagonal can be significant since $R ; S^\smile$ is always difunctional. In fact, Lemma 4.1 can be strenghtened:

Lemma 4.2. Given a square of mappings in an allegory as drawn above, and existence of the difunctional closure of $P^\smile ; Q$, we have $P ; R = Q ; S$ iff $(P^\smile ; Q)^{\boxtimes} \sqsubseteq R ; S^\smile$.

PROOF: The "if" direction follows immediately from $P^\smile ; Q \sqsubseteq (P^\smile ; Q)^{\boxtimes}$ and the "if" direction of Lemma 4.1.

For "only if", assume $P ; R = Q ; S$. Then $P^\smile ; Q \sqsubseteq R ; S^\smile$ by Lemma 4.1, and

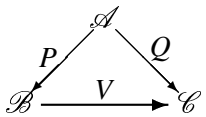
$$\begin{array}{lcll} R ; S^\smile ; Q^\smile ; P ; P^\smile ; Q & = & R ; R^\smile ; P^\smile ; P ; P^\smile ; Q & \text{commutativity} \\ & = & R ; R^\smile ; P^\smile ; Q & P \text{ unival.} \\ & = & R ; S^\smile ; Q^\smile ; Q & \text{commutativity} \\ & \sqsubseteq & R ; S^\smile & Q \text{ unival.} \end{array}$$

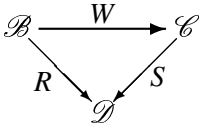
By left-induction for difunctional closure we therefore have $(P^\smile ; Q)^{\boxtimes} \sqsubseteq R ; S^\smile$. □

Producing the result span of a pullback (respectively the result co-span of a pushout) from the horizontal diagonal alone is, in some sense, a generalisation of Freyd and Scedrov's splitting of idempotents; [Kah04] contains more discussion of this aspect.

Definition 4.3. [FS90, 2.14] In an allegory, let a morphism $V : \mathscr{B} \to \mathscr{C}$ be given. The span $\mathscr{B} \xleftarrow{P} \mathscr{A} \xrightarrow{Q} \mathscr{C}$ of *mappings* P and Q is called a *tabulation of* V iff the following equations hold:

$$P^\smile ; Q = V \qquad P ; P^\smile \sqcap Q ; Q^\smile = \mathbb{I}_{\mathscr{A}} \ . \qquad \square$$





Definition 4.4. [Kah04] In a collagory, let a morphism $W : \mathscr{B} \to \mathscr{C}$ be given. The co-span $\mathscr{B} \xrightarrow{R} \mathscr{D} \xleftarrow{S} \mathscr{C}$ of *mappings* R and S is called a *co-tabulation of* W iff the following equations hold:

$$R ; S^\smile = W \qquad R^\smile ; R \sqcup S^\smile ; S = \mathbb{I}_{\mathscr{D}} \ . \qquad \square$$

The first equation implies $W ; W^\smile ; W = R ; S^\smile ; S ; R^\smile ; R ; S^\smile \sqsubseteq R ; S^\smile = W$ (using univalence of R and S), so if W has a co-tabulation, it has to be difunctional.

Furthermore, from univalence of R and S we also obtain the lax cocone conditions $R^\smile ; W = R^\smile ; R ; S^\smile \sqsubseteq S^\smile$ and $W ; S = R ; S^\smile ; S \sqsubseteq R$.

The following equivalent characterisations provided by [Kah04] have the advantage that they are fully equational, without the implicit inclusions in the mapping conditions. This frequently facilitates calculations. Note that $\mathbb{I} \sqcap V ; V^\smile = \mathsf{dom}\, V$; we use the expanded form to emphasise the duality.

Proposition 4.5. In an allegory, the span $\mathscr{B} \xleftarrow{P} \mathscr{A} \xrightarrow{Q} \mathscr{C}$ is a tabulation of $V : \mathscr{B} \to \mathscr{C}$ if and only if the following equations hold:

$$P^\smile ; Q = V \qquad \begin{array}{rcl} P^\smile ; P & = & \mathbb{I} \sqcap V ; V^\smile \\ Q^\smile ; Q & = & \mathbb{I} \sqcap V^\smile ; V \end{array} \qquad P ; P^\smile \sqcap Q ; Q^\smile = \mathbb{I}_{\mathscr{A}} \ . \qquad \square$$

Proposition 4.6. In a collagory, the co-span $\mathscr{B} \xrightarrow{R} \mathscr{D} \xleftarrow{S} \mathscr{C}$ is a co-tabulation of $W : \mathscr{B} \to \mathscr{C}$ iff the following equations hold:

$$R ; S^\smile = W \qquad \begin{array}{rcl} R ; R^\smile & = & \mathbb{I} \sqcup W ; W^\smile \\ S ; S^\smile & = & \mathbb{I} \sqcup W^\smile ; W \end{array} \qquad R^\smile ; R \sqcup S^\smile ; S = \mathbb{I}_{\mathscr{D}} \ . \qquad \square$$

Definition 4.7. If an allegory has a tabulation for each morphism, we call it *tabular*.

If a collagory has a co-tabulation for each morphism, we call it *co-tabular*, and if it is furthermore tabular, we call it *bi-tabular*. $\square$

Tabulations in an allegory are unique up to isomorphism (this uses the modal rule), and include the following special cases:

- In a tabulation of a sub-identity, both tabulation morphisms are the induced *sub-object* injection [FS90, 2.145].
- We can define a *direct product* of $\mathscr{A}$ and $\mathscr{B}$ to be a tabulation of a $\mathbb{T}_{\mathscr{A},\mathscr{B}}$, provided that greatest morphism exists. The resulting direct product definition differs from that of [SS93] in extending naturally to "empty" objects (e.g., empty sets) by not demanding surjectivity of the projections, but only $\pi^\smile ; \pi = \mathsf{dom}\, \mathbb{T}_{\mathscr{A},\mathscr{B}}$ and $\rho^\smile ; \rho = \mathsf{ran}\, \mathbb{T}_{\mathscr{A},\mathscr{B}}$.
- If a co-span $\mathscr{B} \xrightarrow{R} \mathscr{D} \xleftarrow{S} \mathscr{C}$ of mappings is given, then each tabulation of $R ; S^\smile$ (there might be none) is a *pullback* in $\mathsf{Map}\,\mathbf{A}$ [FS90, 2.147].
  For a tabular allegory $\mathbf{A}$, this implies that each pullback in $\mathsf{Map}\,\mathbf{A}$ is isomorphic to a tabulation, and therefore is itself a tabulation. However, if $\mathbf{A}$ is not tabular, then a co-span $\mathscr{B} \xrightarrow{R} \mathscr{D} \xleftarrow{S} \mathscr{C}$ of mappings for which no tabulation of $R ; S^\smile$ exists may still have a pullback in $\mathsf{Map}\,\mathbf{A}$, which then cannot be a tabulation.

If an allegory is known to have all direct products and subobjects, then these can be used to construct a tabulation for each morphism.

In a collagory, we have the following special cases of co-tabulations, dual to the special tabulations above:

- In a co-tabulation of an equivalence relation, both R and S are the induced *quotient* projections.
- We can define a *direct sum* of $\mathscr{A}$ and $\mathscr{B}$ to be a co-tabulation of $\mathbb{\perp}_{\mathscr{A},\mathscr{B}}$, if that least morphism exists.
- If a span $\mathscr{B} \xleftarrow{P} \mathscr{A} \xrightarrow{Q} \mathscr{C}$ of mappings is given, and the difunctional closure $W := (P^\smile ; Q)^{\boxplus}$ exists then each co-tabulation of W (there might be none) is a *pushout* in $\mathsf{Map}\,\mathbf{A}$ [Kah09a].
  The situation is, except for the addition of the difunctional closure, perfectly dual to the situation for pullbacks described above: For a co-tabular collagory $\mathbf{C}$, each pushout in $\mathsf{Map}\,\mathbf{C}$ is isomorphic to a co-tabulation, and therefore is itself a co-tabulation. However, if $\mathbf{C}$ is not co-tabular, then a span $\mathscr{B} \xleftarrow{P} \mathscr{A} \xrightarrow{Q} \mathscr{C}$ of mappings for which no co-tabulation of $(P^\smile ; Q)^{\boxplus}$ exists may still have a pushout in $\mathsf{Map}\,\mathbf{C}$, which then cannot be a co-tabulation.

If direct sums and quotients are available, then a co-tabulation can be constructed for each difunctional morphism.

A co-tabulation for a difunctional closure $Z^{\boxplus}$ satisfies the following equations:

$$R;S^{\smile} = Z^{\boxplus} \qquad R;R^{\smile} = Z^{\Rightarrow} \qquad S;S^{\smile} = Z^{\Leftarrow} \qquad R^{\smile};R \sqcup S^{\smile};S = \mathbb{I}_{\mathscr{D}} \ .$$

This was introduced as a *gluing for* the morphism Z in [Kah01]. Kawahara is the first to have characterised pushouts relation-algebraically in essentially this way [Kaw90]; he used relation-algebraic operations on relations arising in toposes.

Convention 4.8. For a square of morphisms as drawn at the beginning of this section, we say that

- it *is a tabulation* iff $\mathscr{B} \xleftarrow{P} \mathscr{A} \xrightarrow{Q} \mathscr{C}$ is a tabulation for $R;S^{\smile}$,
- it *is a (direct) co-tabulation* iff $\mathscr{B} \xrightarrow{R} \mathscr{D} \xleftarrow{S} \mathscr{C}$ is a co-tabulation for $P^{\smile};Q$,
- it *is a gluing* iff $\mathscr{B} \xrightarrow{R} \mathscr{D} \xleftarrow{S} \mathscr{C}$ is a gluing for $P^{\smile};Q$, that is, if it is a co-tabulation for $(P^{\smile};Q)^{\boxplus}$. □

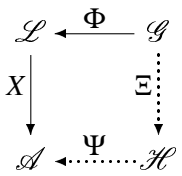
5 The Gluing Condition in Collagories

We can now state a relational variant of the gluing condition, first introduced by Kawahara [Kaw90]:

Definition 5.1. Let two morphisms[1] $\Phi : \mathscr{G} \to \mathscr{L}$ and $X : \mathscr{L} \to \mathscr{A}$ in a collagory with pseudo-complements on subidentities be given.[2]

- We say that the *identification condition* holds iff $X;X^{\smile} \sqsubseteq \mathbb{I} \sqcup (\mathsf{ran}\,\Phi);X;X^{\smile};\mathsf{ran}\,\Phi$.
- We say that the *dangling condition* holds iff $\mathsf{ran}\,X \sqcup (\mathsf{ran}\,X \to \mathsf{ran}\,(\Phi;X)) = \mathbb{I}$. □

The proofs that the gluing condition is sufficient for the existence of a pushout complement [Kaw90], and that injectivity of Φ is sufficient for unambiguity of the pushout complement [Kah01] carry over to the collagory setting, but are outside the scope of this paper.



Another related condition is important in the context of the single-pushout approach:

Definition 5.2. In an allegory, we call X *conflict-free for* Φ iff $\mathsf{ran}\,(\Phi;X;X^{\smile}) \sqsubseteq \mathsf{ran}\,\Phi$. □

For a node-and-edges-level formulation of conflict-freeness it is well-known that the induced single-pushout squares have a total embedding of the right-hand side into the application graph [Löw90, Cor. 3.18.5]. The component-free formulation above was first given in [Kah01], where it is also shown (Thm. 5.4.11) that a restricting derivation step for a conflict-free redex produces a pushout of partial functions.

---

[1] Note that "X" is a capital "χ".

[2] *Pseudo-complements* are residuation of meet in lower semilattice categories; where pseudo-complements exist, we denote the pseudo-complement or R with respect to S as $R \to S$, and we have:

$$X \sqcap R \sqsubseteq S \qquad \Leftrightarrow \qquad X \sqsubseteq (R \to S)$$

For example, the pseudo-complement of a subgraph R of a graph G with respect to another subgraph S consists of all nodes of G that are in S or not in R, and all edges in S or not in R that are also nor incident with nodes in R. Intuitively, $R \to S$ therefore is G with the parts or R outside S removed, and then also all dangling edges removed.

6 Co-tabulations as Bicolimits and Lax Colimits

Ordered categories are a simple example of 2-categories and bicategories: between two morphisms there is at most one two-cell, and there is a two-cell between two morphisms $R, S : \mathscr{A} \to \mathscr{B}$ iff $R \sqsubseteq S$. Therefore, there is an invertible two-cell between R and S if and only if $R = S$.

6.1 OC-Colimits: Bicolimits in Ordered Categories

The general notion of bicolimits takes as its point of departure a *diagram* defined via a functor from a category. We introduce a specialised variant of the definition used in [HS09] by restricting our attention to ordered categories.

Definition 6.1. Given a category $\mathbf{C}$, an (index) category $\mathbf{J}$, a functor $\mathbf{D} : \mathbf{J} \to \mathbf{C}$ defining a diagram, and an object $\mathscr{D}$, a *cocone* η from $\mathbf{D}$ to $\mathscr{D}$ consists of a morphism $\eta_{\mathscr{A}} : \mathbf{D}\mathscr{A} \to \mathscr{D}$ in $\mathbf{C}$ for each object $\mathscr{A}$ of $\mathbf{J}$, satisfying the following *cocone commutativity* condition:

$$\mathbf{D}F ; \eta_{\mathscr{B}} = \eta_{\mathscr{A}} \qquad \text{for each morphism } F : \mathscr{A} \to \mathscr{B} \text{ in } \mathbf{J}. \qquad \square$$

Definition 6.2. Given an ordered category $\mathbf{C}$, an (index) category $\mathbf{J}$, and a functor $\mathbf{D} : \mathbf{J} \to \mathbf{C}$, an *OC-colimit of* $\mathbf{D}$ is given by an object $\mathscr{D}$ of $\mathbf{C}$, and a cocone η from $\mathbf{D}$ to $\mathscr{D}$, satisfying the following conditions:

1. *factorisation:* for any other object $\mathscr{D}'$ of $\mathbf{C}$ with cocone κ from $\mathbf{D}$ to $\mathscr{D}'$, there is a morphism $h : \mathscr{D} \to \mathscr{D}'$ in $\mathbf{C}$ with
$$\eta_{\mathscr{A}} ; h = \kappa_{\mathscr{A}} \qquad \text{for each object } \mathscr{A} \text{ in } \mathbf{J}.$$
2. *isotony:* for any other object $\mathscr{D}'$ of $\mathbf{C}$ and any two morphisms $h, h' : \mathscr{D} \to \mathscr{D}'$, if $\eta_{\mathscr{A}} ; h \sqsubseteq \eta_{\mathscr{A}} ; h'$ for all objects $\mathscr{A}$ in $\mathbf{J}$, then $h \sqsubseteq h'$. $\square$

OC-colimits are unique up to isomorphism.

6.2 Lax Colimits in OCCs

For lax cocones, we only need the concept of lax functor, which differs from the functor concept in that a *lax functor* $\mathbf{D}$ only needs to satisfy $\mathbb{I}_{\mathbf{D}\mathscr{A}} \sqsubseteq \mathbf{D}\mathbb{I}_{\mathscr{A}}$ and $(\mathbf{D}f) ; (\mathbf{D}g) \sqsubseteq \mathbf{D}(f ; g)$, see, e.g., [Stu05, Sect. 8, p. 37ff]. Again, we provide specialised definition of lax cocones and lax colimits for the ordered category case:

Definition 6.3. Given an *ordered* category $\mathbf{C}$, an (index) category $\mathbf{J}$, a lax functor $\mathbf{D} : \mathbf{J} \to \mathbf{C}$ defining a diagram, and an object $\mathscr{D}$, a *lax cocone* η from $\mathbf{D}$ to $\mathscr{D}$ consists of a morphism $\eta_{\mathscr{A}} : \mathbf{D}\mathscr{A} \to \mathscr{D}$ in $\mathbf{C}$ for each object $\mathscr{A}$ of $\mathbf{J}$, satisfying the following *cocone subcommutativity* condition:

$$\mathbf{D}F ; \eta_{\mathscr{B}} \sqsubseteq \eta_{\mathscr{A}} \qquad \text{for each morphism } F : \mathscr{A} \to \mathscr{B} \text{ in } \mathbf{J}. \qquad \square$$

Definition 6.4. Given an ordered category $\mathbf{C}$, an (index) category $\mathbf{J}$, and a lax functor $\mathbf{D} : \mathbf{J} \to \mathbf{C}$, a *lax colimit of* $\mathbf{D}$ is given by an object $\mathscr{D}$ of $\mathbf{C}$, and a lax cocone η from $\mathbf{D}$ to $\mathscr{D}$ satisfying the following conditions

1. *factorisation:* for any object $\mathscr{D}'$ of $\mathbf{C}$ with lax cocone κ from $\mathbf{D}$ to $\mathscr{D}'$, there is a morphism $U : \mathscr{D} \to \mathscr{D}'$ in $\mathbf{C}$ with
$$\eta_{\mathscr{A}} ; U = \kappa_{\mathscr{A}} \qquad \text{for each object } \mathscr{A} \text{ in } \mathbf{J},$$

2. *isotony:* for any object $\mathscr{D}'$ of $\mathbf{C}$ and any two morphisms $U, U' : \mathscr{D} \to \mathscr{D}'$, if $\eta_{\mathscr{A}} ; U \sqsubseteq \eta_{\mathscr{A}} ; U'$ for each object $\mathscr{A}$ in $\mathbf{J}$, then $U \sqsubseteq U'$. □

Lax colimits are unique up to isomorphism, too.

We now add the converse operator to our consideration of lax colimits, and when we use "$\bullet \to \bullet$" to denote an OCC, that OCC has the homset from the first object $\mathscr{A}$ to the second, different object $\mathscr{B}$ contain exactly one morphism, say F, from $\mathscr{A}$ to $\mathscr{B}$. As an OCC, it needs to also have $F^\smile$, which will be the only morphism from $\mathscr{B}$ to $\mathscr{A}$. Since in this OCC, also $F ; F^\smile ; F$ needs to exist as a morphism from $\mathscr{A}$ to $\mathscr{B}$, it has to be equal to F, which therefore is difunctional.

If a lax functor $\mathbf{D}$ maps $F : \mathscr{A} \to \mathscr{B}$ to $W : \mathscr{A}' \to \mathscr{B}'$, then

$$W ; W^\smile ; W = \mathbf{D}F ; (\mathbf{D}F)^\smile ; \mathbf{D}F \sqsubseteq \mathbf{D}(F ; F^\smile ; F) = \mathbf{D}F = W \ ,$$

so it can map $\mathscr{F}$ only to difunctional morphisms.

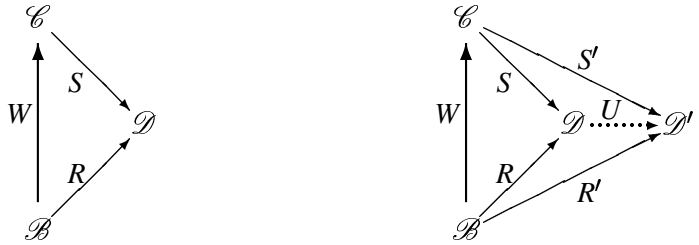
Furthermore, if, for a lax cocone, its source $\mathbf{J}$ is considered as an OCC, this implies that for each morphism $F : \mathscr{A} \to \mathscr{B}$ in $\mathbf{J}$, also the converse morphism $F^\smile : \mathscr{B} \to \mathscr{A}$ needs to be considered. Such a lax cocone therefore automatically has to satisfy both the following conditions:

$$\left.\begin{array}{rcl} \mathbf{D}F ; \eta_{\mathscr{B}} & \sqsubseteq & \eta_{\mathscr{A}} \\ (\mathbf{D}F)^\smile ; \eta_{\mathscr{A}} & \sqsubseteq & \eta_{\mathscr{B}} \end{array}\right\} \quad \text{for each morphism } F : \mathscr{A} \to \mathscr{B} \text{ in } \mathbf{J}.$$

Convention 6.5. Given a morphism $W : \mathscr{B} \to \mathscr{C}$ in the OCC $\mathbf{C}$, we will frequently identify W with the functor $\mathbf{D}$ mapping the single morphism explicitly mentioned in the OCC $\bullet \to \bullet$ to W.

(Since we are dealing with an OCC, that morphism also has a converse, which then must be mapped to $W^\smile$.) □

A lax cocone from W to $\mathscr{D}$ therefore is a cospan $\mathscr{B} \xrightarrow{R} \mathscr{D} \xleftarrow{S} \mathscr{C}$ satisfying $W ; S \sqsubseteq R$ and $W^\smile ; R \sqsubseteq S$.



We explicitly state the definition of resulting special case of lax colimits:

Definition 6.6. An *OCC-colimit* of $W : \mathscr{B} \to \mathscr{C}$ in the OCC $\mathbf{C}$ is a lax cocone $\mathscr{B} \xrightarrow{R} \mathscr{D} \xleftarrow{S} \mathscr{C}$ from W to $\mathscr{D}$ (with $W ; S \sqsubseteq R$ and $W^\smile ; R \sqsubseteq S$) satisfying the following conditions:

1. *factorisation:* for any object $\mathscr{D}'$ of $\mathbf{C}$ with lax cocone $\mathscr{B} \xrightarrow{R'} \mathscr{D}' \xleftarrow{S'} \mathscr{C}$ from W to $\mathscr{D}'$, there is a morphism $U : \mathscr{D} \to \mathscr{D}'$ in $\mathbf{C}$ with $R ; U = R'$ and $S ; U = S'$;
2. *isotony:* for any object $\mathscr{D}'$ of $\mathbf{C}$ and any two morphisms $U, U' : \mathscr{D} \to \mathscr{D}'$, if $R ; U \sqsubseteq R ; U'$ and $S ; U \sqsubseteq S ; U'$, then $U \sqsubseteq U'$. □

The crucial aspect of the following theorem (proof in [Kah10]) is that it connects the respecive O*-limits for spans $\mathscr{B} \xleftarrow{P} \mathscr{A} \xrightarrow{Q} \mathscr{C}$ of mappings with those for the single difunctional morphisms $(P^\smile ; Q)^{\boxplus}$ (which do not need to be mappings).

Theorem 6.7. If a span $\mathscr{B} \xleftarrow{P} \mathscr{A} \xrightarrow{Q} \mathscr{C}$ of mappings in a collagory is given, then a cospan $\mathscr{B} \xrightarrow{R} \mathscr{D} \xleftarrow{S} \mathscr{C}$ is an OCC-colimit for $(P^\smile ; Q)^{\boxplus}$ iff it is an OC-pushout (i.e., OC-colimit for a span) for $\mathscr{B} \xleftarrow{P} \mathscr{A} \xrightarrow{Q} \mathscr{C}$. □

6.3 OCC-Colimits are Co-tabulations

In a collagory $\mathbf{C}$ that is not co-tabular, categorical pushouts in $\mathsf{Map}\,\mathbf{C}$ are not necessarily gluings — the pushout conditions establish no connection between mappings and other morphisms, and pathological cases cannot be excluded.

However, OC-colimits and OCC-colimits do establish the necessary connections; one direction is easy to see (details in [Kah10]):

Theorem 6.8. If a cospan $\mathscr{B} \xrightarrow{R} \mathscr{D} \xleftarrow{S} \mathscr{C}$ in a collagory is a co-tabulation of $W : \mathscr{B} \to \mathscr{C}$, then it is also an OCC-colimit for W. □

We now show that all OCC-colimits (of necessarily difunctional morphisms) are in fact co-tabulations. The proof needs to rely on the lax colimit properties, and therefore needs to use appropriate lax cocones constructed from the morphisms known to exist for a given OCC-colimit. The following lemma already follows this pattern:

Lemma 6.9. If, in an allegory, $\mathscr{B} \xrightarrow{R} \mathscr{D} \xleftarrow{S} \mathscr{C}$ is an OCC-colimit for W, then

$$\begin{array}{rclcrcl} W^\smile ; R & = & S ; \mathsf{ran} R & \qquad & W^\smile ; R ; R^\smile & = & S ; R^\smile \\ W ; S & = & R ; \mathsf{ran} S & & W ; S ; S^\smile & = & R ; S^\smile \end{array}$$

PROOF: Let $R_0 = W ; S$ and $S_0 = S$. This defines a lax cocone $\mathscr{B} \xrightarrow{R_0} \mathscr{D} \xleftarrow{S_0} \mathscr{C}$ from W to $\mathscr{D}$, since:

$$\begin{array}{rcl} W^\smile ; R_0 & = & W^\smile ; W ; S \sqsubseteq W^\smile ; R \sqsubseteq S = S_0 \ ; \\ W ; S_0 & = & W ; S = R_0 \ . \end{array}$$

Then factorisation gives us a $U_0 : \mathscr{D} \to \mathscr{D}$ such that $R_0 = W ; S = R ; U_0$ and $S_0 = S = S ; U_0$. Since $R ; U_0 = W ; S \sqsubseteq R = R ; \mathbb{I}_{\mathscr{D}}$ and $S ; U_0 = S \sqsubseteq S ; \mathbb{I}_{\mathscr{D}}$, isotony gives us $U_0 \sqsubseteq \mathbb{I}_{\mathscr{D}}$.

So U_0 is a sub-identity, and $S = S ; U_0$ implies $\mathsf{ran} S \sqsubseteq U_0$. Since composition of sub-identities is meet, we obtain the following (which implies $U_0 = \mathsf{ran} S$):

$$W ; S = W ; S ; \mathsf{ran} S = R ; U_0 ; \mathsf{ran} S = R ; \mathsf{ran} S$$

Analogously, $W^\smile ; R = S ; \mathsf{ran} R$ also holds, and these further imply

$$W^\smile ; R ; R^\smile = S ; R^\smile \qquad \text{and} \qquad W ; S ; S^\smile = R ; S^\smile \ .$$ □

Lemma 6.9 does not use difunctionality of W, and implies:

$$\begin{array}{rcl} W^\smile ; W ; W^\smile ; R & = & W^\smile ; W ; S ; \mathsf{ran} R = W^\smile ; R ; \mathsf{ran} S ; \mathsf{ran} R \\ & = & W^\smile ; R ; \mathsf{ran} S = S ; \mathsf{ran} R ; \mathsf{ran} S = S ; \mathsf{ran} R = W^\smile ; R \end{array}$$

and, analogously, $W ; W^\smile ; W ; S = W ; S$. Therefore, even with a weaker concept of OCC-colimit, we would still have, in some sense, "almost-difunctionality" of W.

Lemma 6.9 did use allegory properties (for sub-identities); to show the opposite inclusion to Theorem 6.8 we need full collagories (detailed proof in [Kah10]):

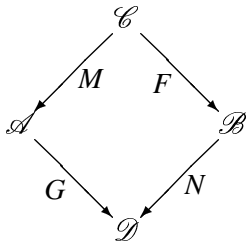
Theorem 6.10. If, in a collagory, $W : \mathscr{B} \to \mathscr{C}$ is a difunctional morphism and $\mathscr{B} \xrightarrow{R} \mathscr{D} \xleftarrow{S} \mathscr{C}$ is an OCC-colimit for W, then it is also a co-tabulation for W. □

In summary, we have shown in Theorem 6.7 that OC-pushouts (i.e., OC-colimits) of a span are the same as OCC-colimits of the difunctional closure of the composition across that span. Furthermore, OCC-colimits of difunctional morphisms are the same as co-tabulations, as shown in Theorems 6.8 and 6.10.

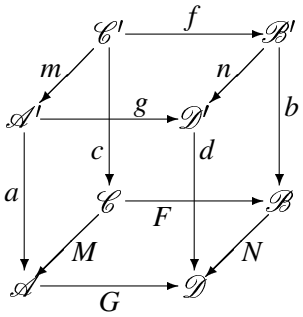
7 Van Kampen Squares in Collagories

Adhesive categories as a more specific setting for double-pushout graph rewriting have been introduced by Lack and Sobociński [LS04, LS05]; the following two definitions are taken from there:

Definition 7.1. A *van Kampen square* (i) is a pushout which satisfies the following condition: given a commutative cube (ii) of which (i) forms the bottom face and the back faces are pullbacks (where $\mathscr{C}$ is considered to be in the back), the front faces are pullbacks if and only if the top face is a pushout.



(i)



(ii) □

Definition 7.2. A category $\mathbf{C}$ is said to be *adhesive* if

1. $\mathbf{C}$ has pushouts along monomorphisms;
2. $\mathbf{C}$ has pullbacks;
3. pushouts along monomorphisms are van Kampen squares. □

For more concise formulations, we define:

Definition 7.3. A *van Kampen setup* in a collagory $\mathbf{C}$ for a square as in Def. 7.1(i) is a commuting cube in $\mathsf{Map}\,\mathbf{C}$ as in Def. 7.1(ii) where the bottom square is a gluing and the two back squares are tabulations. □

In [Kah09b], the following two lemmas were only shown for co-tabulations (i.e., assuming that $M^\smile;F$ is difunctional, and also of $m^\smile;f$ where it is assumed to be a gluing), not for general gluings. In [Kah10], we show the following significantly strengthened versions.

Lemma 7.4. In a collagory, if the front squares of a van Kampen setup are tabulations, then the top square is a gluing. If furthermore $M^\smile;F$ is difunctional, then $m^\smile;f$ is difunctional, too. □

Lemma 7.5. In a van Kampen setup where the top square is a gluing, the front squares are tabulations iff the following holds:

$$m;(m^\smile;f)^{\boxplus};f^\smile \sqcap c;c^\smile \sqsubseteq \mathbb{I}_{\mathscr{C}'}$$ □

The condition here is equivalent to the following inclusion in the lattice of equivalences on $\mathscr{C}'$:

$$(m;m^\smile \vee f;f^\smile) \wedge c;c^\smile = \mathbb{I}_{\mathscr{C}'}$$

Since equivalence lattices are not necessarily distributive, we cannot derive this from the tabulation equations $m;m^\smile \sqcap c;c^\smile = \mathbb{I}_{\mathscr{C}'}$ and $f;f^\smile \sqcap c;c^\smile = \mathbb{I}_{\mathscr{C}'}$.

From Lemmas 7.4 and 7.5, we also directly obtain a characterisation of van Kampen squares in bitabular collagories:

Theorem 7.6. A gluing square (as in Def. 7.1(i)) in a bitabular collagory is van Kampen iff all its van Kampen setups (as in Def. 7.3) where the top square is a gluing satisfy the following:

$$m;(m^\smile;f)^{\boxplus};f^\smile \sqcap c;c^\smile \sqsubseteq \mathbb{I}_{\mathscr{C}'} \qquad \square$$

The bitabularity condition could be weakened, but even then, this characterisation theorem is still very different from the appropriate diagram instance of Heindel and Sobociński's characterisation theorem [HS09, Theorem 22], due to the fact that, by assuming a gluing, we already restricted ourselves to "well-behaved" pushouts.

Our theorem also stays more in the typical relation-algebraic spirit: instead of Heindel and Sobociński's condition "a colimit exists", we have a local inclusion to check. The universal quantification this is embedded in is essentially the same as in [HS09, Theorem 22].

An interesting question is whether there is a useful characterisation that employs a local condition only on the candidate square, beyond injectivity of one M and F, as used in the definition of adhesive categories.

First we observe (proof in [Kah10]):

Lemma 7.7. In a van Kampen setup where $M;M^\smile \sqcap F;F^\smile \sqsubseteq \mathbb{I}_{\mathscr{C}}$, the following hold:

1. $f;f^\smile \sqcap m;m^\smile;c;c^\smile \sqsubseteq \mathbb{I}_{\mathscr{C}'}$
2. $c;c^\smile \sqcap m;m^\smile;f;f^\smile \sqsubseteq \mathbb{I}_{\mathscr{C}'}$ $\square$

Injectivity of M makes $M^\smile;F$ difunctional and also enforces injectivity of m and therewith difunctionality of $m^\smile;f$.

In the general case, however, we have seen above that difunctionality of $m^\smile;f$ requires not only difunctionality of $M^\smile;F$, but also the front tabulation conditions.

This failure of difunctionality propagation can be understood as coming from the fact that in the difunctionality inclusion $M^\smile;F;F^\smile;M;M^\smile;F \sqsubseteq M^\smile;F$, the right-hand side passes through a "$\mathscr{C}$ element" that may be distinct from the three "$\mathscr{C}$ elements" of the left-hand side.

This distinct "$\mathscr{C}$ element" gives rise to a "$\mathscr{C}'$ element" that is, in the absence of the front tabulation conditions, determined only up to $c;c^\smile$.

One way to avoid this unwanted factor is to specify that in any chain diagram documenting $M;M^\smile;F;F^\smile;M;M^\smile$, the fourth (i.e., last) $\mathscr{C}$ element needs to be one of the previous three $\mathscr{C}$ elements. Referring to so many elements simultaneously in a relation-algebraic way requires direct products — we use π and ρ as the projections. The following is one formulation of this condition:

$$M;M^\smile;(\pi^\smile \sqcap F;F^\smile;M;M^\smile;\rho^\smile) \sqsubseteq M;M^\smile;(\pi^\smile \sqcap (F;F^\smile \sqcup M;M^\smile);\rho^\smile)$$

However, it is not hard to see that this is equivalent to the following, much simpler condition:

$$F;F^\smile;M;M^\smile \sqsubseteq F;F^\smile \sqcup M;M^\smile$$

This is obviously satisfied if one of M and F is injective. It can also be strengthened to an equality, since M and F are both total. This implies symmetry:

$$F;F^\smile;M;M^\smile = F;F^\smile \sqcup M;M^\smile = M;M^\smile;F;F^\smile$$

and, furthermore, difunctionality of $M^\smile;F$:

$$M^\smile;F;F^\smile;M;M^\smile;F = M^\smile;M;M^\smile;F;F^\smile;F = M^\smile;F \quad .$$

Assuming also $M;M^\smile \sqcap F;F^\smile \sqsubseteq \mathbb{I}_{\mathscr{C}}$, we obtain $f;f^\smile;m;m^\smile = f;f^\smile \sqcup m;m^\smile$:

$$\begin{array}{lll} & f;f^\smile;m;m^\smile & \\ = & f;f^\smile;m;m^\smile \sqcap c;F;F^\smile;M;M^\smile;c^\smile & \\ \sqsubseteq & f;f^\smile;m;m^\smile \sqcap c;(F;F^\smile \sqcup M;M^\smile);c^\smile & \text{assumption} \\ = & f;f^\smile;m;m^\smile \sqcap (c;c^\smile;f;f^\smile \sqcup c;c^\smile;m;m^\smile) & \\ = & (f;f^\smile;m;m^\smile \sqcap c;c^\smile;f;f^\smile) \sqcup (f;f^\smile;m;m^\smile \sqcap c;c^\smile;m;m^\smile) & \\ \sqsubseteq & f;f^\smile \sqcup m;m^\smile & \text{Lemma 7.7} \end{array}$$

Therefore, $m^\smile;f$ is difunctional, too, and together with Lemma 7.7 we obtain

$$m;(m^\smile;f)^{\boxplus};f^\smile \sqcap c;c^\smile = m;m^\smile;f;f^\smile \sqcap c;c^\smile \sqsubseteq \mathbb{I}_{\mathscr{C}'} \quad .$$

Altogether we have shown the following:

Theorem 7.8. In the category $\mathsf{Map}\,\mathbf{C}$ of maps over a bi-tabular collagory $\mathbf{C}$, pushouts for spans $\mathscr{A} \xleftarrow{M} \mathscr{C} \xrightarrow{F} \mathscr{B}$ that satisfy also

$$F;F^\smile \sqcap M;M^\smile \sqsubseteq \mathbb{I}_{\mathscr{C}} \qquad \text{and} \qquad F;F^\smile;M;M^\smile \sqsubseteq F;F^\smile \sqcup M;M^\smile$$

are van Kampen squares. □

Both inclusions can be strengthened to equalities, and since the second condition implies difunctionality, both together imply that such pushouts are also pullbacks.

8 Conclusion

We have shown that, in collagories, lax colimits of single morphisms are the same as co-tabulations, and bicolimits of spans (bipushouts) are the same as gluings. Furthermore, the move from a span $\mathscr{B} \xleftarrow{P} \mathscr{A} \xrightarrow{Q} \mathscr{C}$ to the difunctional closure of $P^\smile;Q$ preserves both kinds of colimits. (The opposite move could be achieved via a tabulation, and may still deserve to be spelt out.)

We also strengthened our previous results about the two implications involved in van Kampen squares from difunctional spans to arbitrary spans, extracted a precise relation-algebraic condition for van Kampen squares in collagories, and gave a new, purely local sufficient condition for van Kampen squares that is more general than the "pushouts along monomorphisms" used in adhesive categories.

These two results together with the fact that the equational characterisation of co-tabulations enables a nice, calculational proof style make a strong case to employ collagories as a convenient basis for theoretical investigations of graph structure transformations. In addition, relation-algebraic formulations and reasoning are accessible to a wide audience due to the fact that in the intuitive special case of *Rel*, they can be understood as Boolean matrix operations.

Future investigations will explore how these new conditions for van Kampen squares can be combined with the different variations of adhesive categories in a collagory setting, including the quasiadhesive categories of [LS05], and their applications.

Bibliography

[CMR+97] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, M. Löwe. Algebraic Approaches to Graph Transformation, Part I: Basic Concepts and Double Pushout Approach. In Rozenberg (ed.), *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 1: Foundations*. Chapter 3, pp. 163–245. World Scientific, Singapore, 1997.

[EPPH06] H. Ehrig, J. Padberg, U. Prange, A. Habel. Adhesive High-Level Replacement Systems: A New Categorical Framework for Graph Transformation. *Fund. Inform.* 74(1):1–29, 2006. http://iospress.metapress.com/content/f89c8ba4nbeq1xc4/

[FS90] P. J. Freyd, A. Scedrov. *Categories, Allegories*. North-Holland Mathematical Library 39. North-Holland, Amsterdam, 1990.

[HS09] T. Heindel, P. Sobociński. Van Kampen Colimits as Bicolimits in Span. In Kurz and Tarlecki (eds.), *Algebra and Coalgebra in Computer Science, CALCO 2009*. LNCS. Springer, Sept. 2009. (To appear).

[Kah01] W. Kahl. A Relation-Algebraic Approach to Graph Structure Transformation. 2001. Habil. Thesis, Fakultät für Informatik, Univ. der Bundeswehr München, Techn. Report 2002-03, http://sqrl.mcmaster.ca/~kahl/Publications/RelRew/.

[Kah04] W. Kahl. Refactoring Heterogeneous Relation Algebras around Ordered Categories and Converse. *J. Relational Methods in Comp. Sci.* 1:277–313, 2004.

[Kah09a] W. Kahl. Collagories for Relational Adhesive Rewriting. In Berghammer et al. (eds.), *Relations and Kleene Algebra in Computer Science, RelMiCS/AKA 2009*. LNCS 5827, pp. 211–226. Springer, 2009.
doi:10.1007/978-3-642-04639-1

[Kah09b] W. Kahl. Collagories for Relational Adhesive Rewriting. SQRL Report 56, Software Quality Research Laboratory, Department of Computing and Software, McMaster University, July 2009. In: http://sqrl.mcmaster.ca/sqrl_reports.html (Superseded by [Kah10]).

[Kah10] W. Kahl. Collagory Notes, Version 1. SQRL Report 57, Software Quality Research Laboratory, Department of Computing and Software, McMaster University, Mar. 2010. In: http://sqrl.mcmaster.ca/sqrl_reports.html. (This report supersedes [Kah09b]).

[Kaw90] Y. Kawahara. Pushout-Complements and Basic Concepts of Grammars in Toposes. *Theoretical Computer Science* 77:267–289, 1990.

[Löw90] M. Löwe. Algebraic Approach to Graph Transformation Based on Single Pushout Derivations. Technical report 90/05, TU Berlin, 1990.

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

[LS05] S. Lack, P. Sobociński. Adhesive and quasiadhesive categories. *RAIRO Inform. Théor. Appl.* 39(3):511–545, 2005.
http://www.numdam.org/item?id=ITA_2005__39_3_511_0

[SS93] G. Schmidt, T. Ströhlein. *Relations and Graphs, Discrete Mathematics for Computer Scientists*. EATCS-Monographs on Theoretical Computer Science. Springer, 1993.

[Stu05] I. Stubbe. Categorical Structures Enriched in a Quantaloid: Categories, Distributors and Functors. *Theory and Appl. of Categories* 14(1):1–45, 2005.
http://tac.mta.ca/tac/volumes/14/1/14-01abs.html