Electronic Communications of the EASST
Volume 10 (2008)

Proceedings of the Seventh International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2008)

Composing control flow and formula rules for computing on grids[1]

P. Bottoni and N. Mirenkov and Y. Watanobe and R. Yoshioka

15 pages

Guest Editors: Claudia Ermel, Reiko Heckel, Juan de Lara
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

[1] Work done while the first author was working in Aizu University as adjunct professor.

Composing control flow and formula rules for computing on grids[2]

P. Bottoni[1] and N. Mirenkov[2] and Y. Watanobe[2] and R. Yoshioka[2]

[1] bottoni@di.uniroma1.it Dep. of Computer Science, "Sapienza" Univ. of Rome, Italy

[2](nikmir,yutaka,rentaro)@u-aizu.ac.jp Dep. of Computer Software, Univ. of Aizu, Japan

Abstract: We define computation on grids as the composition, through pushout constructions, of control flows, carried across adjacency relations between grid cells, with formulas updating the value of some attribute. The approach is based on the identification of a subcategory of attributed typed graphs suitable to the definition of pushouts on grids, and is illustrated in the context of the Cyberfilm visual language.

Keywords: Grids, Control flow rules, DPO

1 Introduction

Graphs have been long proposed as a universal formalism for describing the structure of system configurations and to support computational specifications of the transformations they may undergo. Moving from this common ground, the areas of graph transformations and graph algorithms have taken two divergent, possibly complementary paths.

On the one hand, graph transformations propose a declarative approach to computation based on the iteration of local modifications to the graph structure, so as to define a language of admissible graph configurations, each depicting a possible state of the system being modelled.

On the other hand, algorithms on graphs exploit procedural definitions of visits to the graph structure, usually to extract some global property of it. In many cases, graph transformations – typically performed by enriching the graph with additional features such as types [CMR96], attributes [MW93, HKT02, dBE+07] or control structures on rule application [KK99, SWZ99, BKPT00] – are capable of replicating many relevant features of the algorithmic approach.

However, the general approach to graph transformations – based on the search for a subgraph isomorphism between the antecedent of a rule and the host graph under scrutiny – is not optimal for spatially organized structures, such as grids (in any number of dimensions), trees, or pyramids [YM02, WMYM08], as the inherent non-determinism of the matching process fails to take advantage of the existence of privileged relations among elements, and of orders for their visit.

We propose to reconcile the use of graph transformation as a general computational framework with the existence of some spatial structure on the host graph. To this end, we combine a suite of meta-models for diagrammatic languages – defining the possible spatial relations among identifiable elements [BG04], their transformation semantics [BLG07], and the relations between the two [dGB07] – with a form of algebraic composition of rules in the framework of the Double Pushout Approach to graph rewriting.

The proposal is applied to the *Cyberfilm* visual environment, which provides the user with iconic representations of computational flows on spatial structures. These representations are

[2] Work done while the first author was working in Aizu University as adjunct professor.

arranged as sequences of frames highlighting the set of nodes which at each step contribute to the production of a new result [WMYM08]. In a separate view, the formulas defining the computations can be defined, thus allowing their reuse according to different control flows. In particular, we focus on bidimensional grids on which several control flows can be defined, and use a categorical construction to provide a formal treatment of the composition of control flow and computational formulas.

In the rest of the paper, after related work in Section 2, we provide background on graph transformations and the adopted metamodels in Section 3. Section 4 introduces the categories on grids needed to define control flow rules in Section 5. Finally, Section 6 shows how to compose formulas and control flows, before drawing conclusions in Section 7.

2 Related Work

Spatial structures, such as those defined by grids or trees, have been the subject of many studies from the algorithmic point of view, in particular as regards the identification of paths with particular properties over them [IPS82]. From the algebraic point of view, trees have been studied as representations of computational structures, such as terms [HP95] or abstract syntaxes [Mos94], whereas images, rather than grids, have been studied in relation to the sets of languages definable on them [GR97]. The translation morphism discussed in this paper may be seen as an analogous of the ”positional overlapping” operation for images [BL07].

The technique for composing control flows and formulas differs from the notion of (local) application of rules to rules in [Par94], based on finding a match from a rule component to a component of another rule, as well as from that of *action pattern* in [BLG07], where a pattern is matched to the right-hand side of a rule to produce a rule whose effects conform to the pattern. A construction analogous to the one here is in [TB94], exploiting common subrules to identify possible agreements on a host graph and construct amalgamated versions of the rules. Although we can also use this notion to find agreements between rule application, we are mainly interested here in the construction of new rules from rules defined on different graph types.

Finally, we point out a similarity with notions of modularity and Viewpoints [GEMT00], proposed as a way to modeling a system through the integration of partial models. However, we combine different aspects of the behaviour of the system into an integrated specification, rather than considering behavioral and structural aspects together. The approach to coordination proposed in [AFGK02] is also based on pushouts (actually colimits), to allow separation of concerns when defining different aspects of a program behaviour.

3 Background: Metamodels and Graph Transformations

According to the metamodel for diagrammatic languages presented in [BG04], and shown in Figure 1, a *diagram* is composed of *identifiable elements* among which significant *spatial relations* exist. A whole diagram is itself an identifiable element, with global properties. An identifiable element is a recognizable unit in the language, associated with a graphical representation defined by a *complex graphic element*, composed in turn of one or more *graphic elements*, each possessing some *attach zone*, representing the geometrical support for spatial relations. The existence of

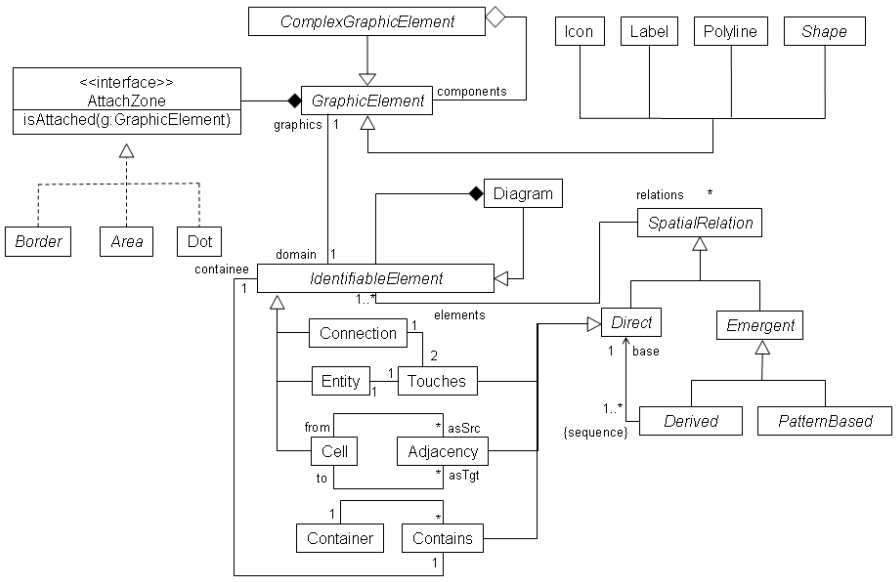

Figure 1: The overall metamodel for diagrammatic languages.

a relation is assessed via a predicate `isAttached()` implemented by each zone. Symmetries may exist between spatial relations. Two relations σ and ρ are tied by a symmetry if there is a size-preserving diagram transformation changing all instances of ρ into instances of σ.

Specializations of these abstract types define language families. For example, in connection-based languages, an `Entity` acts as an endrole for `Connection` elements, while the significant relation is `Touches`, determined by the coincidence of a *dot* at the end of a connection with a point on the *border* of an entity. In this paper we are interested in languages based on `Adjacency`, which indicates a class of relations between `Cells` of regular shape tessellating the plan, and whose *borders* overlap for a finite segment. According to the type of tessellation, cells may entertain various adjacency relations, typically possessing symmetric companions, such as in `Left` and `Right` adjacency for regular arrangements of rectangles.

Based on this metamodel, we can represent diagrams as attributed typed graphs, where nodes are elements of classes in the metamodel and edges are instances of the associations.

Formally, a *type graph* is a construct $TG = (N_T, E_T, s^T, t^T)$ with N_T and E_T sets of node and edge types. $s^T : E_T \to N_T$ and $t^T : E_T \to N_T$ define the *source* and *target* node types for each edge type. A typed graph on TG is a graph $G = (N, E, s, t)$ with a graph morphism $type : G \to TG$ composed of $type_N : N \to N_T$ and $type_E : E \to E_T$, s.t. $type_N(s(e)) = s^T(type_E(e))$ and $type_N(t(e)) = t^T(type_E(e))$. Type graphs with node inheritance exploit a pair $TGI = (TG, I)$, where $I = (N_I, E_I, s^I, t^I)$ is a node inheritance graph, with $N_I = N_T$, i.e. I has the same nodes as TG, but its edges are the inheritance relations. The inheritance *clan* of a node n is the set of all its children nodes (including n itself): $clan(n) = \{n' \in N_I | \exists \text{ path } n' \to^* n \text{ in } I\} \subseteq N_I$.

Typed attributed graphs are typed graphs with additional *data* nodes and *attribute edges* (nodes of G are now called *object nodes*). A type graph TG has a set Δ of *data type nodes* and a set A of *attribute type edges*, denoting the domains of the attribute nodes and the set of attributes associ-

ated with nodes, together with functions $\sigma_A : N_T \to \mathscr{P}(A)$, defining the attributes for a given type, and $\tau_A : A \to \Delta$, defining the admissible domain for each attribute. These elements define a *type graph with attributes* TG_A. A typed attributed graph on TG_A is a construct $(TG, G, N_\Delta, E_A, s_A, t_A)$, where TG and G are as before, N_Δ is the set of data nodes, coinciding with the disjoint union of the domains of attributes, and E_A is the set of *attribute edges*, from object nodes to data nodes. Edges are typed on A and associate object nodes with the values of its attributes. $s_A : E_A \to N$ and $t_A : E_A \to N_\Delta$ define the valuation of attributes for a given node, coherently with σ_A and τ_A. We represent data nodes as typed items and distinguish them from object nodes through a dotted contour, following the convention proposed in Example 8.5 of [EEPT06].

Attributed typed graphs form the adhesive HLR category [LS04] $\mathbf{AGraph_{ATG}}$, so that transformations can be expressed through Double Pushout (DPO) derivations [EEPT06], in which rules are spans $L \leftarrow K \rightarrow R$ and K defines the part which is left unchanged by the rule application. We also exploit application conditions, as shown in Section 4.

4 Categories on Grids

Rectangular grids are regular arrangements of cells according to symmetrical pairs of *vertical* and *horizontal* adjacency relations, with *nrow* rows and *ncol* columns, conforming to the family of adjacency-based diagrammatic languages depicted in Figure 2. Hence, nodes are instances of `Cell`, with a position given by their *row* and *col* attributes and boolean values to distinguish border cells. Adjacency relations can be of four types, with the obvious constraints on their pairing. Moreover, there exists a set of additional constraints stating that a grid has to form a rectangle (i.e. its top and bottom borders must have the same number of elements, as must its left and right borders), and all border elements are adjacent to three other cells except the four corner elements, adjacent to two. Mirror and rotation symmetries exists between pairs of adjacency relations.

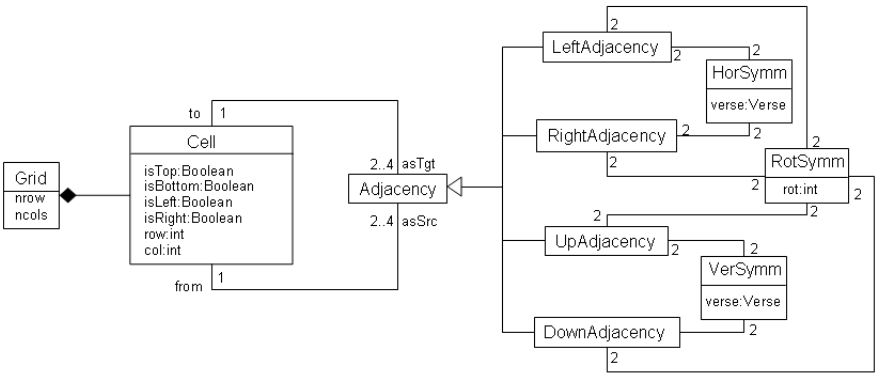

Figure 2: The family of bidimensional grids.

Grids thus give rise to a subcategory $\mathbf{AGrid_T}$ of $\mathbf{AGraph_{ATG}}$, whose morphisms are composed of a *structural* part, involving nodes of type `Cell` and `Adjacency`, and a *data* part involving

attributes. The structural part uses *translations* as morphisms. A translation exists from a grid G_1 to a grid G_2 if G_2 is such that an isomorphism exists from G_1 to a subset of its cells, preserving its connectivity and the relative directions. A translation is uniquely determined by the position of the image of the upper left corner (or any other cell) of the original grid in the context of the target grid. Figure 3 shows the composition of two translations, where the highlighted rectangles show the new positions of the original grid. We assume that translations occur only rightwards and downwards. The pairs (r,c) labeling the morphisms indicate the offsets at which the nodes of the original grid are found in the new grid. The size of G_2 is at least equal to that of G_1. The identity morphism is the translation $(0,0)$ from a grid into itself, and morphism composition is the vectorial sum of the translations. We now study the subcategory $\mathbf{TGrid_T}$, obtained by taking the structural part of $\mathbf{AGrid_T}$, i.e. maintaining the type information, but forgetting attributes.

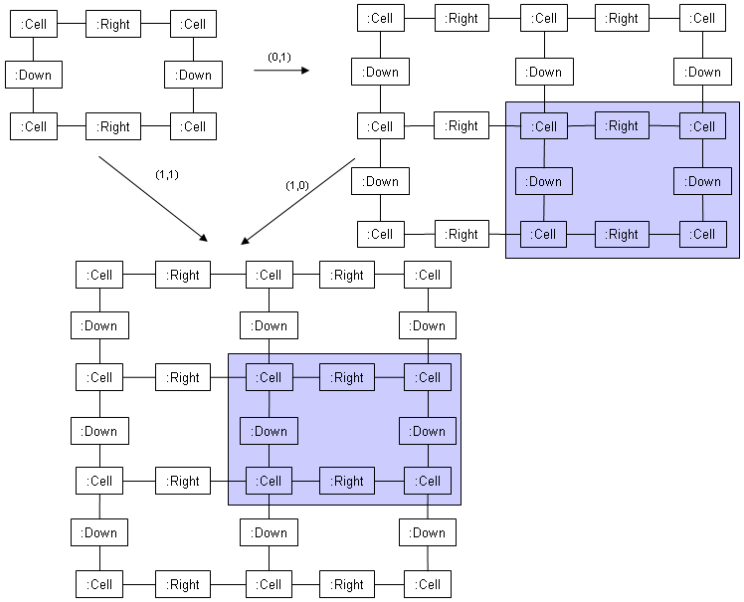

Figure 3: Translation morphism and composition.

In $\mathbf{TGrid_T}$, a pushout $G_1 \xrightarrow{p_1} P \xleftarrow{p_2} G_2$ for a span $G_1 \xleftarrow{t_1} G \xrightarrow{t_2} G_2$ between two grids can be constructed in the same way as the pushout in **Graph** if and only if one of the following is true:

(1) either t_1 or t_2 is an identity;
(2a) t_1 has a label of the form $(r_1,0)$ and $G_1.ncol == G.ncol$ AND
(2b) t_2 has a label of the form $(0,c_1)$ and $G_2.nrow == G.nrow$;
(3a) t_1 has a label of the form $(0,c_2)$ and $G_1.nrow == G.nrow$ AND
(3b) t_2 has a label of the form $(r_2,0)$ and $G_2.ncol == G.ncol$.

Then, P has size $(max(G1.nrow,G2.nrow),\ max(G1.ncol,G2.ncol))$; morphisms $p_1\colon G_1 \to P$ and $p_2\colon G_2 \to P$ are labeled by $(max(r_1,r_2)-r_1,\ max(c_1,c_2)-c_1)$, and $(max(r_1,r_2)-r_2,\ max(c_1,c_2)-c_2)$, respectively, so that parallel arrows have the same label (see Figure 4). The pushout complement $G \xrightarrow{x_1} C \xrightarrow{x_2} G'$, for the composition $G \xrightarrow{t_1} G_1 \xrightarrow{t_2} G'$, where G is an object of

size (r,c), G_1 of size (r_1,c_1) and G' of size (r',c'), uniquely exists only if t_1 and t_2 satisfy the constraints above, and has size $((r'-r_1)+r,(c'-c_1)+c)$, with x_1 and x_2 labeled as t_1 and t_2.

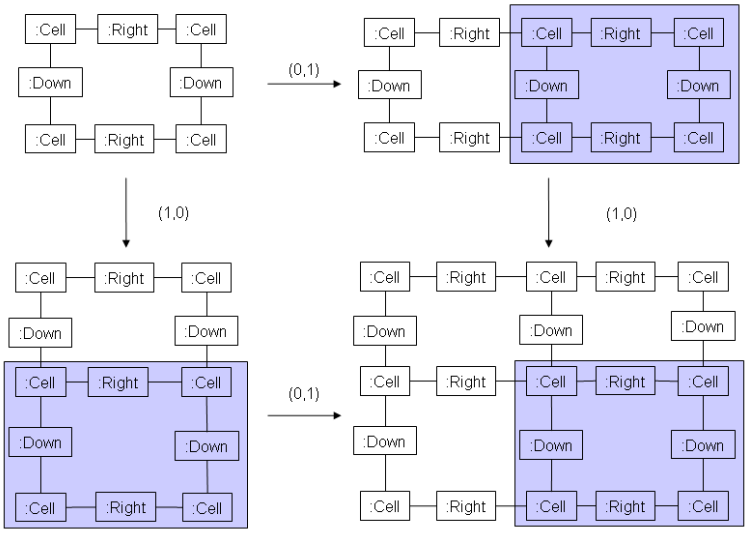

Figure 4: The pushout construction for grids

We can now define *structural* DPO rules in $\mathbf{TGrid_T}$ in accordance to the construction above. In particular, in order to satisfy the the dangling condition, rules are non-deleting (i.e. $L \leftarrow K$ is an identity). The gluing condition, if $K \rightarrow R$ is not an identity, requires border cells of L to be matched to border cells of the host grid G. The pushout complement object D is now always equal to G. Hence, grids can be generated so that the constraints on their rectangular form are maintained through the pushout construction, without having to adopt regulatory mechanisms for rewriting, such as those needed for the so-called Indian grammars: a set of *horizontal* rules is there first used to create the upper row, and then *vertical* rules are applied in parallel to populate the columns [SK74]. The pushout construction can now be lifted in order to consider also attributes. In particular, as typical of attributed graph rewriting, data morphisms are identities (no domain element can be created or deleted). Hence, the effect of a rule can only be the addition of structural nodes and edges and the deletion and creation of attribute edges. Application conditions can be used to describe the relations between values. All grids in $\mathbf{AGrid_T}$ can now be generated by the iterated use of the two rules in Figure 5 and Figure 6, in which identifiers of `Adjacency` nodes indicate their directions and an application condition defines the coordinates of the new cell.

In both cases, we show the classical representation of DPO rules at the top of the figure, and use, at its bottom, a compact notation, already exploited in [dGB07]: the difference between K, L, and R is shown by highlighting the deleted and produced parts with different colours and marking them with tags {`del`} and {`new`}. The elements outside the tagged areas are those belonging to the K component. Note that, differently from [EEPT06], we explicitly show K as presenting isolated data nodes, which are connected to different object nodes in L and R.

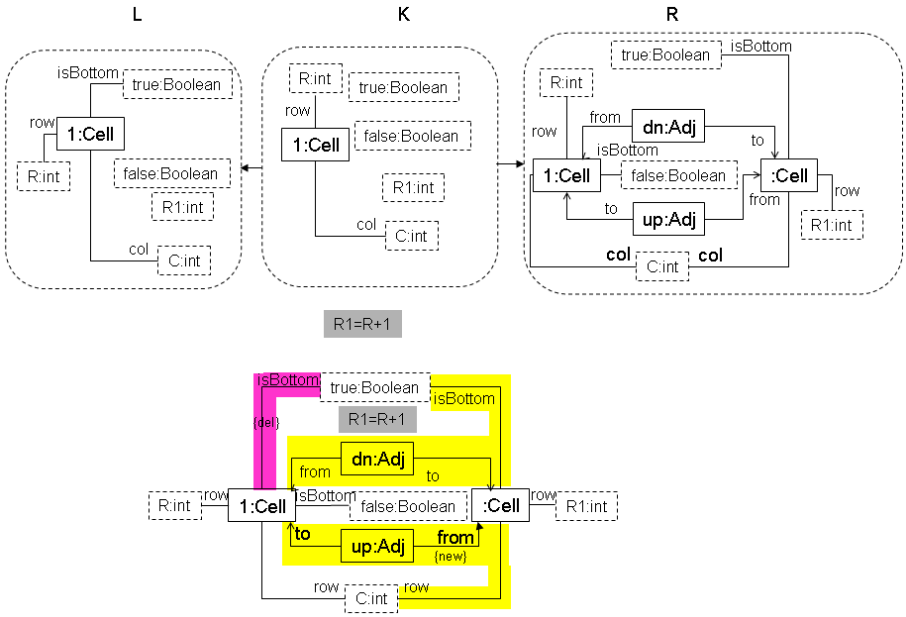

Figure 5: The rule for letting a grid grow horizontally.

5 Control Flow Rules

Figure 7 presents the metamodel triples describing the correspondences induced by assigning to the adjacency relation in the visual representation the semantic meaning of a *carrier*, along which either data or modifications in the *activation state* can travel from and to *active elements*. The left upper part of Figure 7 constitutes the static semantics for the control flow variety on spatial structures, while the right upper part models the data variety, here simplified by considering a simple integer-valued attribute, called `level`. By applying the construction in [dGB07] one can incrementally define the flow structure through triple graph rules which introduce carriers in correspondence with the installation of adjacency relations in specified directions. Hence, one can model the *permeability* of the cell wall to control or data flow. As an example, flows could travel rightwards and leftwards, but not downwards and upwards.

A control flow (*cf*) rule is a DPO rule in $\mathbf{AGraph}_{\mathbf{ATG}}$ with graphs conforming to the left upper part of Figure 7, so that A contains the attribute *state* with values in some finite domain $ActivationState \in \Delta$.

In general, as shown in the rule (in compact form) on the left of Figure 8[3], the activation state may vary during transportation, e.g. a flow can decrease its intensity. The element reached by the flow could have possessed some other activation value and the one from which the flow originated may gain a new one, as defined by application conditions. The basic rule on the right of Figure 8 deals with the case of cells entering an *active* state as the control flow reaches them traveling the grid rightwards from the origin to the destination, while the origin enters a *quiescent* state.

[3] Abbreviations are used for names of values and types.

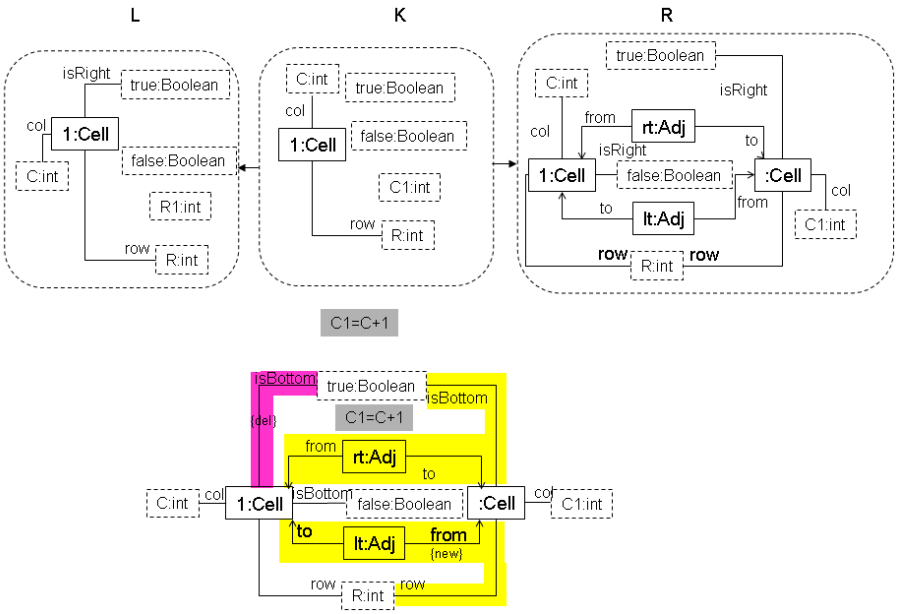

Figure 6: The rule for letting a grid grow vertically.

The carrier identifier indicates the value of its `Direction` attribute. Similar rules are defined for other directions, exploiting rotational and mirror symmetries. *cf*-rules are composed to form more complex ones using a componentwise pushout construction in $\mathbf{AGraph}_{\mathbf{ATG}}$, as shown in Figure 9, where $L \leftarrow K \rightarrow R$ is the maximal intersection of $L_1 \leftarrow K_1 \rightarrow R_1$ and $L_2 \leftarrow K_2 \rightarrow R_2$, all the squares commute and those with curved arrows are pushouts. As an example, directional rules compose through a rule on a single cell passing from the *active* to the *quiescent* state. Figure 10 illustrates the case of the two horizontal movements, while Figure 11 that of one directional and one vertical movement.

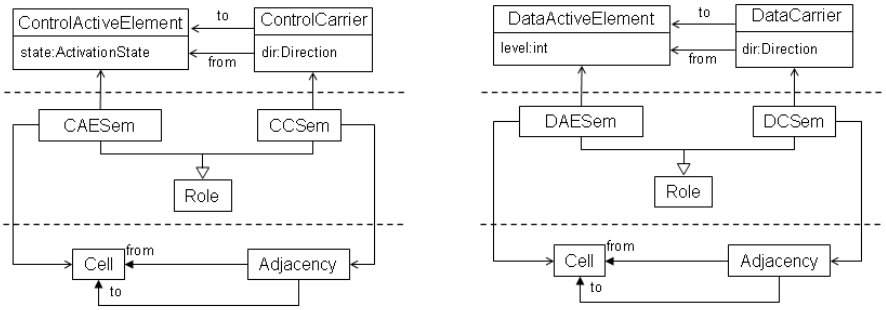

Figure 7: The metamodel triples for control and data flows on grids.

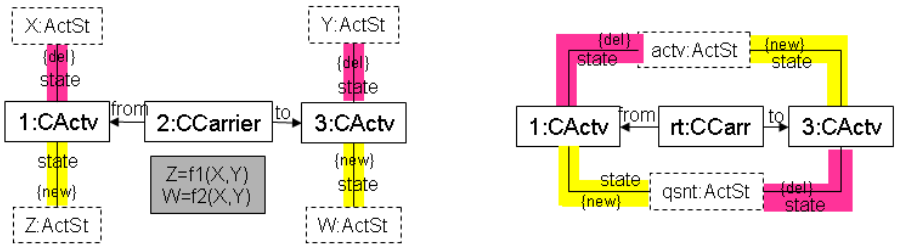

Figure 8: A generic rule for transmission of control flows and a basic rule.

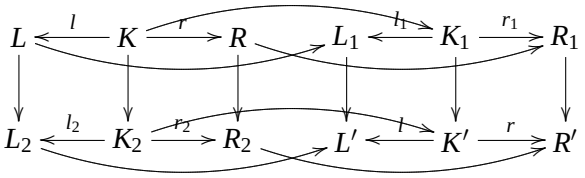

Figure 9: The construction for rule composition.

6 Composing Control Flow and Computation Formulas

We now introduce *data*-rules to specify the transformation of some attribute according to some formula. These are defined on the type graph in the right upper part of Figure 7. Data and *cf*-rules are composed, again with a pushout construction, to produce rules which both apply the formula and propagate the flow, when an active element is reached by the control flow.

The rule in Figure 12 doubles the value of `level`. X and Y are variables to indicate generic instances of an integer. The rules involved in their combination operate on four different types of graphs. The intersection is defined in a type graph where `ActiveElement` abstracts on `ControlActiveElement` and `DataActiveElement` and has no attribute, while the pushout object complies with a type graph formed by taking the quotient of the disjoint union of the two type systems from Figure 7 and identifying the activity types in a `FullActiveElement` type

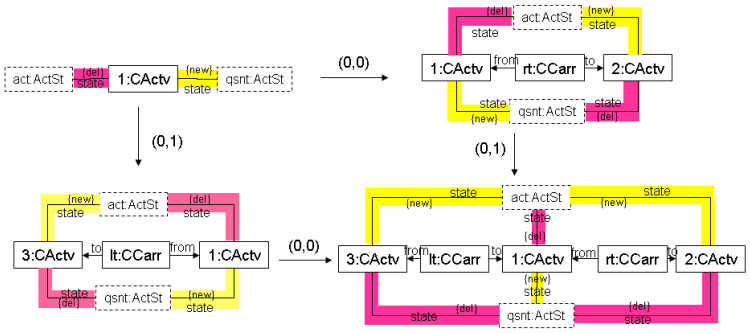

Figure 10: The construction of the rule propagating control flow horizontally in both verses.

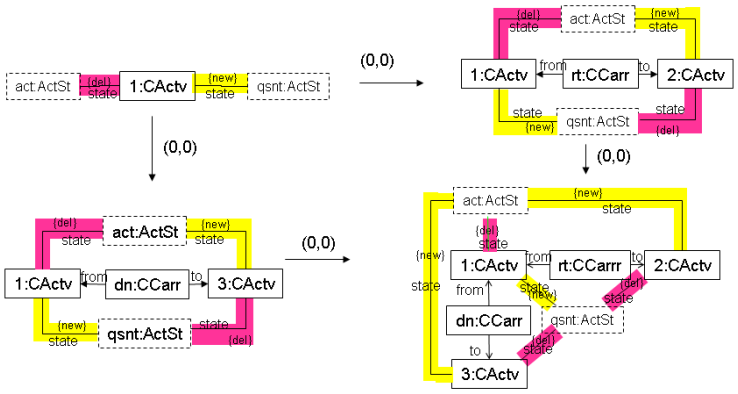

Figure 11: The construction of a bidirectional rule.

(abbreviated in *FActv*). Node morphisms go from less to more specific types.

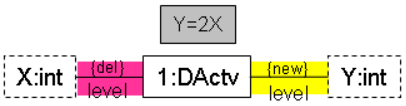

Figure 12: A rule expressing a computational formula.

In Figure 13, the bidirectional rule of Figure 11 is composed with the formula of Figure 12, so that the latter is now evaluated only when the activation front leaves an element in both directions. Using different mappings from the intersection to the *cf*-rule, the formula would be evaluated when the control flow reaches an element from a specific direction. The resulting rule does not specify the `level` values for the other elements. Rules can be applied sequentially or, if they do not conflict on their result, combined to form amalgamated rules to achieve an effect of parallelism [TB94]. As an example, the rule of Figure 13 agrees with itself on any node whose upper and left neighbours are both mapped, by two distinct matches, to the cell identified by 1. Rules can be enriched with parameters and applied via rule expressions to realize complex computations [BKPT00].

6.1 Types of activation in *Cyberfilm*

The *Cyberfilm* language [YM02, WMYM08] provides a collection of predefined control flows, associated with program templates defining the loops realizing them, and with sequences of iconic schemes for an intuitive visualization of the main steps in the execution flow. *Cyberfilm* allows the separated definition of computational formulae and control flow specifications. Hence, the constructions above can be exploited to provide a compositional mechanism for it.

In particular, in the Cyberfilm framework, control flow is defined by the transformation of the

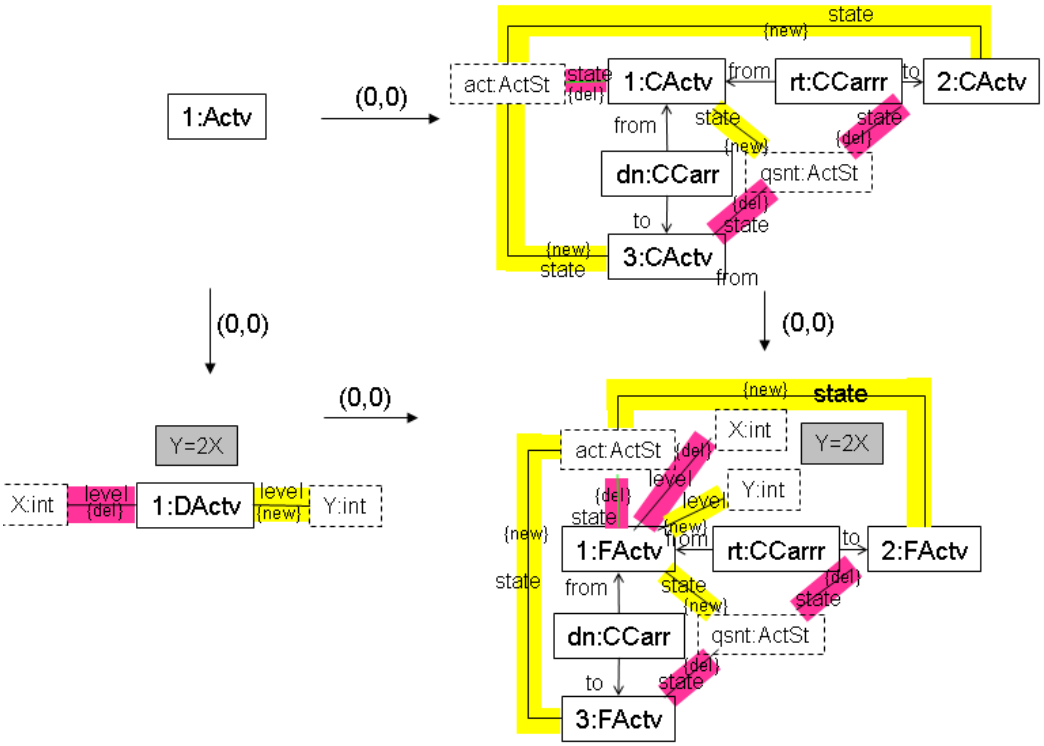

Figure 13: The resulting rule specifying the condition of application.

flashing state of a node Different types of flashing are defined: for example, *full flashing* indicates that the node is able to perform reading and writing operations; *contour flashing* indicates that the node is referenced by other flashing nodes which can perform reading operations, but not change its value; *half flashing* indicates the activity state of an observer which can change the state of other nodes in a global fashion. Other types of flashing are defined, but in this paper we restrict ourselves to the flow of the full and contour flashing, thus interpreting full flashing as an indication that the formula associated with the node can be evaluated to assign a new value to the node, and contour flashing, as the fact that the value of the node is available for formula evaluation by other nodes.

At any time, a cell is in only one possible state. The control flows of the *full* and *contour* flashing can be independent or coordinated. Independent flows can be specified as described in Section 5, whereas coordinated flows require the identification of the conditions under which a cell is able to receive the contributions of other cells.

Figure 14 shows the composition of a coordinated *cf*-rule, for rightward transmission of both *full* and *contour* flows, with a formula rule where an element reads the value of its down neighbour (without changing it), to compute its new level. The upward adjacency relation is mapped, in the formula rule, to a *data carrier* (DC) element, as control and data may flow in different directions, i.e. cells can have different *permeability* to data and control flows.

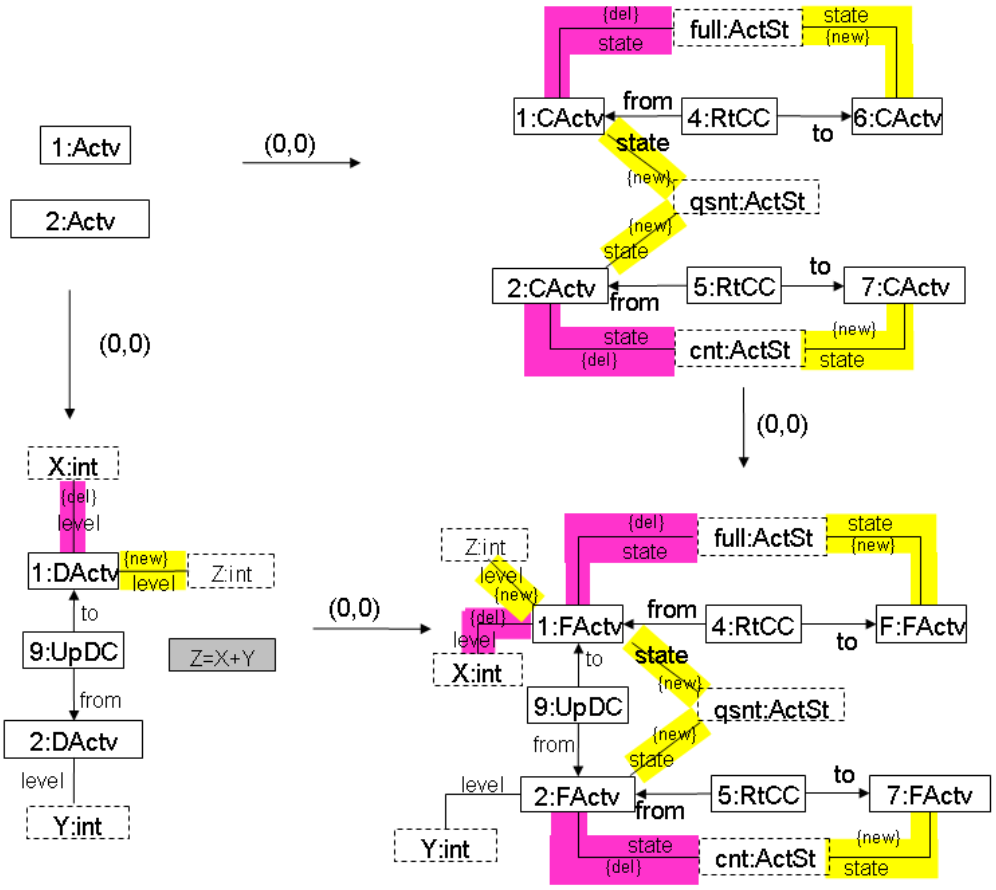

Figure 14: The rule resulting from the coordination of movements of full and contour flashing.

In this case, the *cf*-rule is bound to consider both the *full* and *contour* flows simultaneously. The same effect could be achieved by considering the two flows independently. Figure 15 shows the first step of the relative construction, in which the rightwards movement for the full flashing is combined with the same formula to produce a rule which does not affect the state of the `ControlActiveElement` identified by 2. In Figure 16, the obtained rule is composed with that for rightward movement of contour flashing. While the final effect is the same, the intermediate step could be combined with other movement rules. Several such rules might be defined, for example to propagate the flow across several cells, so that only some elements are activated.

7 Conclusions

We have proposed an approach, based on componentwise pushout of DPO rules in the category of attributed typed graphs, to the specification of computations on grids. The approach allows

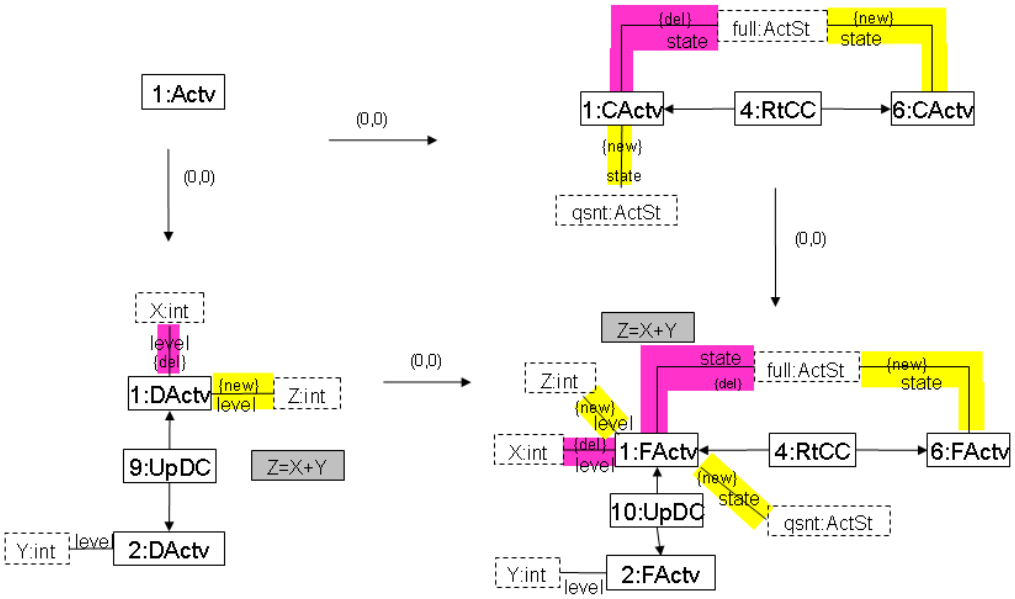

Figure 15: Coordinating movement of the full flashing flow with the formula in Figure 14.

the independent definition of two types of rules, one to specify control flow and the other to specify the actual computations. The construction is symmetrical in control and formula rules, so that it can be flexibly applied starting from either specification. Symmetries between adjacency relations can also be exploited to generate different versions of flows and formulas.

Future work will explore other types of spatial structures, typically trees and pyramids, to define adequate *cf*-rules, also considering the distinction between formula evaluation on control flows reaching or leaving the involved cells, and develop ways of reasoning about the compatibility of independent *cf*-rules (e.g. one for reading and one for writing).

Bibliography

[AFGK02] L. F. Andrade, J. L. Fiadeiro, J. Gouveia, G. Koutsoukos. Separating computation, coordination and configuration. *J. of Software Maintenance* 14(5):353–369, 2002.

[BG04] P. Bottoni, A. Grau. A Suite of Metamodels as a Basis for a Classification of Visual Languages. In *Proc. VL/HCC 2004*. Pp. 83–90. 2004.

[BKPT00] P. Bottoni, M. Koch, F. Parisi Presicce, G. Taentzer. Automatic Consistency Checking and Visualization of OCL Constraints. In *Proc. UML 2000*. Pp. 294–308. 2000.

[BL07] P. Bottoni, A. Labella. Pointed pictures. *Journal of Visual Languages and Computing* 18:523–536, 2007.

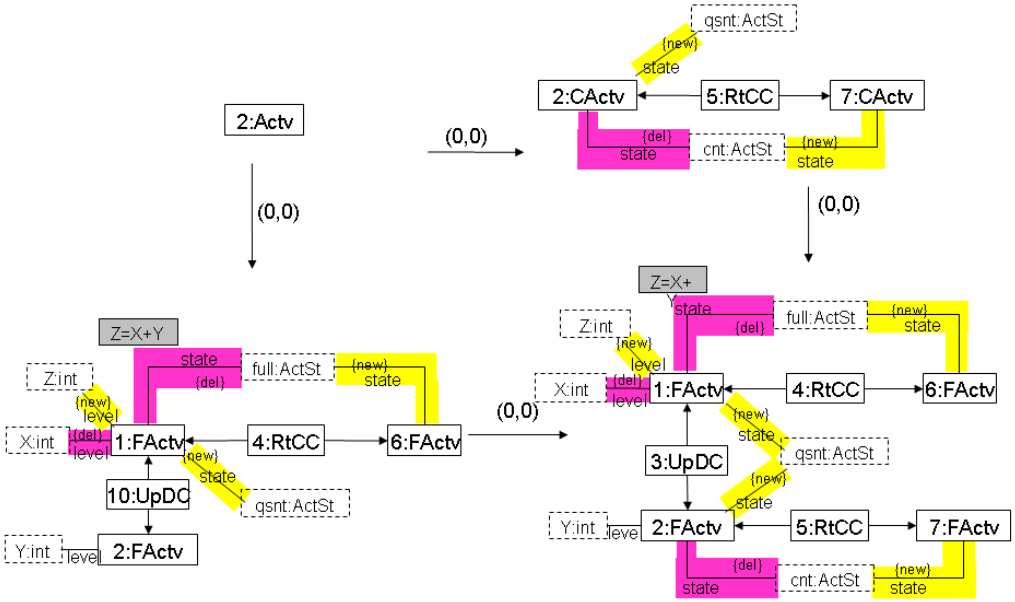

Figure 16: Coordinating movement of the contour flashing flow with the rule of Figure 15.

[BLG07] P. Bottoni, J. de Lara, E. Guerra. Action Patterns for Incremental Specification of Execution Semantics of Visual Languages. In *Proc. VL/HCC 2007*. Pp. 163–170. 2007.

[CMR96] A. Corradini, U. Montanari, F. Rossi. Graph processes. *Fundamenta Informaticae* 26(34):241–265, 1996.

[dBE+07] J. de Lara, R. Bardohl, H. Ehrig, K. Ehrig, U. Prange, G. Taentzer. Attributed graph transformation with node type inheritance. *TCS* 376:139–163, 2007.

[dGB07] J. de Lara, E. Guerra, P. Bottoni. Triple Patterns: Compact Specifications for the Generation of Operational Triple Graph Grammar Rules. In *Proc. GT-VMT'07*. Pp. 81–95. 2007.

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

[GEMT00] M. Goedicke, B. Enders, T. Meyer, G. Taentzer. Towards integration of multiple perspectives by distributed graph transformation. In Nagl et al. (eds.), *Proc. AGTIVE 1999*. Pp. 369–377. 2000.

[GR97] D. Giammarresi, A. Restivo. Two-dimensional languages. In *Handbook of Formal Languages*. Volume III, pp. 215–267. Springer, 1997.

[HKT02] R. Heckel, J. Küster, G. Taentzer. Confluence of Typed Attributed Graph Transformation with Constraints. In *Proc. ICGT 2002*. LNCS 2505, pp. 161–176. 2002.

[HP95] A. Habel, D. Plump. Unification, rewriting, and narrowing on term graphs. *Electr. Notes Theor. Comput. Sci.* 2, 1995.

[IPS82] A. Itai, C. H. Papadimitriou, J. L. Szwarcfiter. Hamilton Paths in Grid Graphs. *SIAM J. Comput.* 11(4):676–686, 1982.

[KK99] H. Kreowski, S. Kuske. Graph Transformation Units with Interleaving Semantics. *Formal Aspects of Computing* 11:690–723, 1999.

[LS04] S. Lack, P. Sobocinski. Adhesive Categories. In Ehrig et al. (eds.), *Proc. FOSSACS 2004*. Pp. 273–288. Springer, 2004.

[Mos94] P. Mosses. *Recent Trends in Data Type Specification*. Chapter Unified algebras and abstract syntax, pp. 280–294. Springer, 1994.

[MW93] M. K. M Löwe, A. Wagner. *Term Graph Rewriting: Theory and Practice*. Chapter An Algebraic Framework for the Transformation of Attributed Graphs, pp. 185–199. John Wiley and Sons Ltd, 1993.

[Par94] F. Parisi Presicce. Transformations of Graph Grammars. In *TAGT*. LNCS 1073, pp. 428–442. 1994.

[SK74] R. Siromoney, K. Krithivasan. Parallel context-free grammars. *Information and Control* 24:155–162, 1974.

[SWZ99] A. Schürr, A. Winter, A. Zündorf. The PROGRES-Approach: Language and Environment. In Ehrig et al. (eds.), *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 2*. Pp. 487–550. World Scientific, 1999.

[TB94] G. Taentzer, M. Beyer. Amalgamated Graph Transformations and Their Use for Specifying AGG. In *Dagstuhl Seminar on Graph Transformations in Computer Science*. LNCS 776, pp. 380–394. Springer, 1994.

[WMYM08] Y. Watanobe, N. N. Mirenkov, R. Yoshioka, O. Monakhov. Filmification of methods: A visual language for graph algorithms. *Journal of Visual Languages and Computing* 19(1):123–150, 2008.

[YM02] R. Yoshioka, N. N. Mirenkov. Visual computing within environment of self-explanatory components. *Soft Computing* 7(1):20–32, 2002.