Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 206 (2008) 133–152
www.elsevier.com/locate/entcs

Using Matrix Graph Grammars for the Analysis of Behavioural Specifications: Sequential and Parallel Independence

Pedro Pablo Pérez Velasco, Juan de Lara

Escuela Politécnica Superior
Universidad Autónoma de Madrid (Spain)

Abstract

In this paper we present a new approach for the analysis of rule-based specification of system dynamics. We model system states as simple digraphs, which can be represented with boolean matrices. Rules modelling the different state changes of the system can also be represented with boolean matrices, and therefore the rewriting is expressed using boolean operations only.
The conditions for sequential independence between pair of rules are well-known in the categorical approaches to graph transformation (e.g. single and double pushout). These conditions state when two rules can be applied in any order yielding the same result. In this paper, we study the concept of sequential independence in our framework, and extend it in order to consider derivations of arbitrary finite length. Instead of studying one-step rule advances, we study independence of rule permutations in sequences of arbitrary finite length. We also analyse the conditions under which a sequence is applicable to a given host graph. We introduce rule composition and give some preliminary results regarding parallel independence. Moreover, we improve our framework making explicit the elements which, if present, disable the application of a rule or a sequence.

Keywords: Graph Transformation, Matrix Graph Grammars, Parallelism, Sequential Independence.

1 Introduction

Graphs are pervasive in many areas of computer science, e.g. to model different kinds of diagrams in software engineering, data structures, or the state space of a dynamical system. Graph transformation is a visual, formal and declarative technique for graph manipulation [4,5]. It is based on the concepts of grammar, rule and derivation. A graph grammar is made of a set of rules – each having graphs in its left and right hand sides (LHS and RHS) – and an initial host graph. If an occurrence (a morphism) of a rule's LHS is found in the host graph, then it can be substituted by the RHS. Graph transformation is becoming increasingly popular, e.g. to specify the operational semantics of diagrammatic languages and visual

[1] Emails: [pedro.perez|jdelara]@uam.es

doi:10.1016/j.entcs.2008.03.079

simulation [8], to express and analyse refactorings or re-designs [9], or for model-to-model transformations [14]. The main advantages of graph grammars with respect to other behavioural specification techniques is that they are a visual, formal and declarative means to express transformations of different kinds of graphs. Different formalizations provide analysis techniques e.g. to study rule independence, confluence or termination (partially) [4,5,11]. The most popular formalizations are based on category theory and include the single [3] and double pushout [2,4] (SPO and DPO).

Graph transformation can be used to model parallel computations in two ways [2]. The first one is using an explicit approach, where a processor is assigned to each process and actions are carried out simultaneously (this is also called truly concurrency). In the second one, processes are modelled by arbitrary interleavings of their actions. These two approaches are related to the notions of parallel and sequential independence. In the latter, two sequences of actions are independent if they can be performed in any order yielding the same result. Sequential and parallel independence have been studied in the categorical approaches for pairs of rules, and conditions have been stated for both of them.

We have recently introduced a formalization of (simple di)graph transformation based on boolean matrix algebra [11,12]. In our approach, the rewriting as well as the analysis techniques can be expressed using boolean matrix operations only. In previous work [11], we introduced some analysis techniques that can be used independently of the host graph. Then, we introduced derivations [12] and how they influence these results. Here we focus on sequential independence, extending it to derivations of arbitrary finite length. Sequential independence for pairs of rules does not extrapolate to sequences of arbitrary length, as sometimes it is possible to advance a rule two or more positions in a derivation, even if the rule is not independent with the following one in the sequence. We also present new results concerning the problem of sequence *applicability*: given a sequence and a host graph, we seek the conditions under which the sequence is applicable to the graph. This is relevant if the sequence should be applied atomically (e.g. when implementing *transactional* properties for rule-based programs).

We also introduce in this paper the notion of rule composition, which allows calculating a single rule able to produce the same result as a rule sequence. Using this concept, we give some preliminary results regarding parallel independence, where we assume that no dangling edges are produced.

Finally, we have improved our approach by making explicit (i.e. representing them as a proper graph) the elements (edges) which if present in the host graph, would prohibit a rule or sequence application. These graphs are the *nihilation matrix* and the *negative initial digraph* respectively, and contain information about potential dangling edges (i.e. edges that would become dangling when deleting certain nodes) and edges that cannot be present as another edge with the same source and target is added by the rule or sequence (simple graphs do not allow more than one edge with same source and target). To the very best of our knowledge, this idea is not present in any other graph transformation approach.

Paper organization. We start with a brief presentation of parallel and sequential independence. Then, Section 3 gives an overview of our *Matrix Graph Grammars*, introducing the new characterization of the *Nihilation* matrix and the new concept of rule composition and sequence compatibility. Section 4 briefly introduces the match, together with some new considerations concerning the handling of dangling edges (that we call *marking*). Section 5 shows the new results for sequence applicability and the new concept of *negative initial digraph*. Section 6 presents the new results for sequential independence for derivations. Section 7 shows new preliminary results concerning parallel independence. Section 8 compares with related research and section 9 ends with the conclusions and future work.

2 Rule Independence

We briefly introduce sequential and parallel independence for SPO and DPO as included in [3]. Parallel independence checks whether two *alternative* direct derivations $H_1 \overset{p_1,m_1}{\Longleftarrow} G \overset{p_2,m_2}{\Longrightarrow} H_2$, are not in conflict (i.e. if each can be applied after the other has been performed, and thus could be applied in parallel) [5]. Sequential independence checks if two *consecutive* direct derivations $G \overset{p_1,m_1}{\Longrightarrow} G \overset{p_2,m_2}{\Longrightarrow} X$ can be swapped.

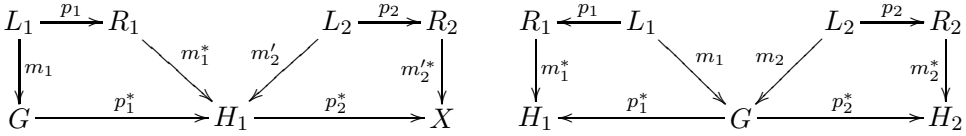


Fig. 1. Sequential (left) and Parallel Independence (right).

Single Pushout. For sequential independence we have $m'_2(L_2) \cap m^*_1(R_1 \setminus p_1(L_1)) = \emptyset$ and $m'_2(L_2 \setminus dom(p_2)) \cap m^*_1(R_1) = \emptyset$, and for parallel independence $m_2(L_2) \cap m_1(L_1 \setminus dom(p_1)) = \emptyset$ and $m_1(L_1) \cap m_2(L_2 \setminus dom(p_2)) = \emptyset$ (see Fig. 1, taken from [3], which we synthesized in Fig.2).

In [3] it is demonstrated that d_1 is sequential independent of d'_2 (written $d_1 \perp d'_2$) iff $\exists\, m_2 : L_2 \to G$ such that $m'_2 = p^*_1 \circ m_2$ and d_1 is weakly parallel independent of derivation d_2 (this condition is known as *weak sequential independence*).

Double Pushout. In DPO, two direct derivations are parallel independent (resp., sequential independent) if all elements in the intersection of both matches (resp., of the comatch of the first derivation and the match of the second) are already gluing items with respect to both transformations. Gluing items of a production p are edges and nodes of its LHS not deleted by p.

3 Matrix Graph Grammars

This section briefly introduces Matrix Graph Grammars (MGGs). Refer to [11] for a more comprehensive presentation. Subsection 3.1 presents the encoding of graphs and rules by means of boolean matrices. Subsection 3.2 studies rule sequences,

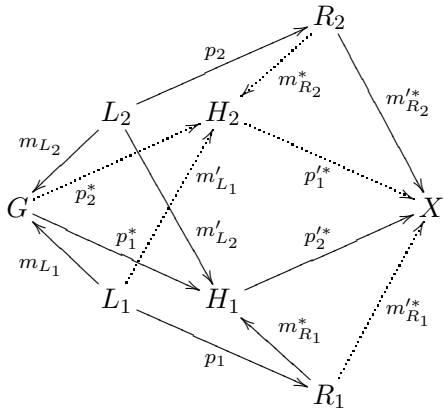
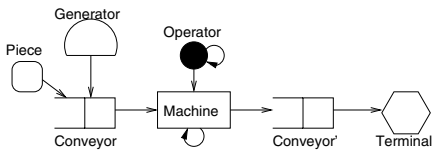


Fig. 2. Sequential and Parallel Independence (synthesized).

and some analysis techniques that can be used independently of the host graph. Subsection 3.3 presents new results concerning rule composition.

3.1 Graphs and Productions

Graphs. We work with simple digraphs, which can be represented as a tuple (M, N) where M is a boolean matrix for edges and N a boolean vector for nodes. The latter is necessary as in the rewriting we can add and delete nodes. Fig. 3(a) shows an example of a graph representing a manufacture system made of a machine, which receives and produces pieces through conveyors. The output conveyor is connected to a terminal element. The machine needs an operator in order to perform its task. Generators produce pieces in conveyors, which have unbounded capacity. Self loops in operators and machines indicate that they are busy.



$$M = \left[\begin{array}{ccccccc|c} 0 & 1 & 0 & 0 & 0 & 0 & 0 & Gen \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & Conv \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & Conv' \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & Mac \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & Ope \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & Ter \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & Piec \end{array}\right], N = \left[\begin{array}{c|c} 1 & Gen \\ 1 & Conv \\ 1 & Conv' \\ 1 & Mac \\ 1 & Ope \\ 1 & Ter \\ 1 & Piec \end{array}\right]$$

Fig. 3. (a) A Simple Digraph Example. (b) Matrix Representation.

Compatibility. Well-formedness of graphs (i.e. no dangling edges) can be checked by verifying the identity $\left\|\left(M \vee M^t\right) \odot \overline{N}\right\|_1 = 0$, with M the edges matrix, $\overline{N}$ the negation of the nodes vector, $\odot$ the boolean matrix product (like the regular matrix product, but with **and** and **or** instead of mutiplication and addition), and $\|\cdot\|_1$ is an operation (a norm, actually) that results in the **or** of all the components of the vector. We call this property *compatibility* [11].

Typing. A type is assigned to each node by a function from the set of nodes $V = |N|$ to a set of types T, $type\colon V \to T$. In Fig. 3, types were represented as

an additional column in the matrices. For edges we use the types of their source and target nodes. The *primas* in the figure allow distinguishing individual elements with same type.

Productions. A production, or rule, $p : L \to R$ is a partial injective function of simple digraphs. Using a *static formulation*, we can represent a rule by two boolean matrices and two vectors $p = \left(L^E, R^E; L^N, R^N\right)$, (where E stands for edges and N for nodes) to characterize the LHS and RHS. The actions performed by a production are addition and deletion of nodes and edges. Therefore, using a *dynamic formulation*, a rule can be represented by $p = \left(L^E, e^E, r^E; L^N, e^N, r^N\right)$, where e^E and e^N are the deletion boolean matrix and vector, while r^E and r^N are the addition boolean matrix and vector. These matrices and vectors have a 1 in the position where the element is to be deleted or added respectively. The output of rule p can be calculated by the boolean formula $R = p(L) = r \vee \overline{e}\, L$, where the formula applies both to nodes and edges. Superindices E and N are omitted if the formula applies to both cases. Moreover, we usually omit the $\wedge$ (**and**) symbol.

Fig. 4 shows a rule and its associated matrices. The rule models the consumption of a piece by a machine. Compatibility of the resulting graph must be ensured, therefore the rule cannot be applied if the machine is already busy, as it would end up with two self loops, which is not allowed in a simple digraph. This restriction of simple digraphs can be useful in this kind of situations, and acts like a built-in negative application condition [4]. Later we will see that the *Nihilation matrix* takes care of this restriction.

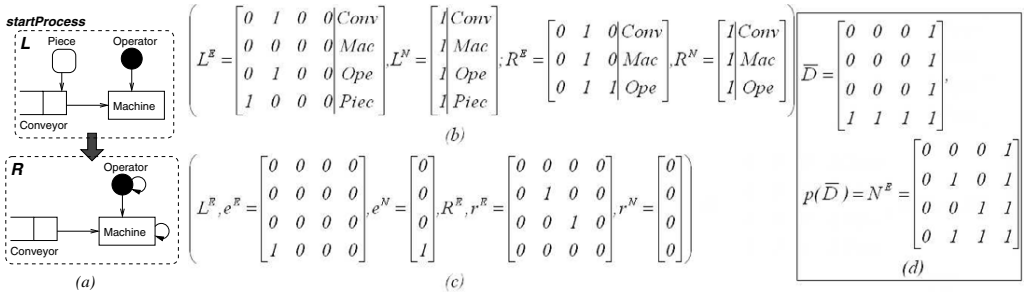


Fig. 4. (a) Rule Example. Static (b) and Dynamic (c) Formulations. (d) Nihilation Matrix

Completion. In order to operate graphs of different sizes, an operation called completion adds extra rows and columns with zeros (to matrices and vectors) and rearranges rows and columns so that the identified edges and nodes of the two graphs match. In the examples, we omit such operation, but assume that matrices are completed when necessary. Later we will operate with the matrices of different productions, which means that we have to select the elements (nodes and edges) of each production which get identified to the same element in the host graph. Thus the completion has to preserve such identifications.

Nihilation Matrix. In order to consider the elements in the host graph that disable a rule application, we extend the notation for rules with a new graph N. Its associated matrix specifies the two kinds of forbidden edges: those incident to nodes which are going to be erased and any edge added by the rule (which cannot

be added twice, since we work with simple digraphs). Notice that N considers only potential dangling edges with source and target in the nodes belonging to L^N.

The concept of rule remains unaltered because we are just making explicit some implicit information. Matrices are derived in the following order: $(L, R) \mapsto (e, r) \mapsto N$. Thus, a rule is *statically* determined by its LHS and RHS $p = (L, R)$, from which it is possible to give a dynamic definition $p = (L; e, r)$, with $e = L\overline{R}$ and $r = R\overline{L}$, to end up with a full specification including its *environmental* behaviour $p = (L, N; e, r)$. Thus, no extra effort is needed from the grammar designer, as matrix N can be automatically calculated as the image by rule p of a certain matrix:

Theorem 3.1 (Nihilation matrix) *Using tensors,* [2] *let* $D = \overline{e^N} \otimes \overline{(e^N)^t}$ *then* $N^E = p\left(\overline{D}\right)$.

Proof. The following matrix specifies potential dangling edges incident to nodes appearing in the LHS of p.

$$\overline{D} = d^i_j = \begin{cases} 1 \text{ if } (e^i)^N = 1 \text{ or } (e^j)^N = 1. \\ 0 \text{ otherwise.} \end{cases}$$

Note that $D = \overline{e^N} \otimes \overline{(e^N)^t}$. Every incident edge to a node that is deleted becomes dangling, except those explicitly deleted by the production. In addition, edges added by the rule cannot be present in the host graph, $N^E = r^E \vee \overline{e^E}\left(\overline{D}\right) = p\left(\overline{D}\right)$. □

Fig. 4(d) shows the nihilation matrix N^E for the example rule. The matrix indicates any dangling edge from the deleted piece (the edge to the conveyor is not indicated as it is explicitly deleted), as well as self-loops in the machine and in the operator. Matrix N^E can be extended to a graph by taking the nodes that appear in the LHS: $N = (N^E, L^N)$. The nihilation matrix should not be confused with the notion of *Negative Application Condition* [4], which is an additional graph specified by the designer (i.e. not derived from the rule) containing extra negative conditions.

3.2 Studying Rule Sequences

Given a collection of productions $\{p_1, \ldots, p_n\}$, $s_n = p_n; p_{n-1}; \ldots; p_1$ defines a sequence (or concatenation) of rules establishing an order in their application, starting with p_1 and ending with p_n (i.e. from right to left). A concatenation is said to be *coherent* if actions carried out by one production do not prevent the application of those coming afterwards. We assume a certain identification of nodes and edges between rules (i.e. matrices have been completed in a certain way and some overlapping of rule elements can occur, which is one of the effects of matches) thus, coherence is calculated with respect to the given identification. Productions can appear more than once in a sequence, even completed in different ways. Next theorem gives the conditions for sequence coherence (see [11] for a complete proof).

[2] Symbol $\otimes$ denotes tensor product, which sums up the covariant and contravariant parts and multiplies every element of the first vector by the whole second vector.

Theorem 3.2 (Sequence Coherence) *Sequence* $s_n = p_n; \ldots; p_1$ *is coherent if* $\bigvee_{i=1}^{n} \left(R_i \bigtriangledown_{i+1}^{n} (\overline{e_x}\, r_y) \vee L_i \bigtriangleup_{1}^{i-1} (e_y\, \overline{r_x})\right) = 0$ *where*

$$\bigtriangleup_{t_0}^{t_1} (F(x,y)) = \bigvee_{y=t_0}^{t_1} \left(\bigwedge_{x=y}^{t_1} (F(x,y)) \right) \tag{1}$$

$$\bigtriangledown_{t_0}^{t_1} (G(x,y)) = \bigvee_{y=t_0}^{t_1} \left(\bigwedge_{x=t_0}^{y} (G(x,y)) \right) \tag{2}$$

Coherence allows the grammar designer to check dependencies between rules, and to realize possible conflicts. The problematic elements are shown as non-zero elements in the resulting matrix.

Fig. 5 shows additional rules for the example. Sequence $s_3 = breakdown; endProcess; startProcess$ (where we have identified nodes with same type across productions) is not coherent as a "1" is obtained in the position corresponding to the self-loops edges of the operator and the machine. This means that both loops are needed in order to execute the given sequence. A possible solution is to have an additional machine and operator. Thus, conflicts detected by coherence may be solved if the initial host graph provides enough edges and nodes (i.e. with a different identification of elements across productions). This is related to the *minimal initial digraph* (MID), which is a graph containing the necessary elements for a sequence to be applicable. Next theorem presents the formula for its calculation.

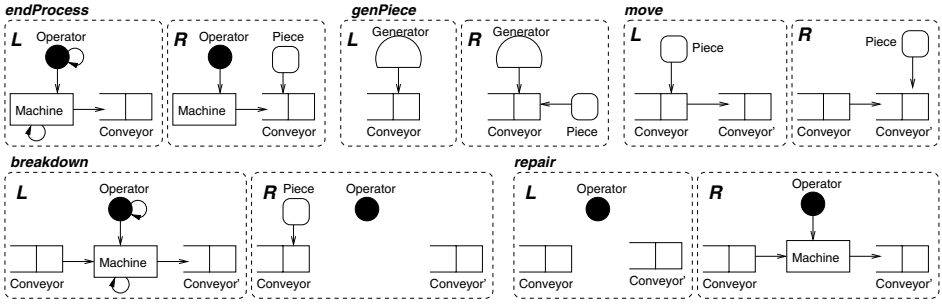


Fig. 5. Additional rules for the example.

Theorem 3.3 (Minimal Initial Digraph) *Given a coherent concatenation of productions* $s_n = p_n; \ldots; p_1$*, its MID is defined by:* $M_n = \bigtriangledown_1^n (\overline{r_x} L_y)$.

Consider sequence $s_3' = startProcess; startProcess; genPiece$, which is not coherent if we identify both operators and machines. Therefore, we need two different machines and two operators, one machine can consume the generated piece, while a different piece is needed for the other machine. For the case of three productions, the formula for the MID expands to $M_3 = (\overline{r}_1 L_1) \vee (\overline{r}_1 L_2)(\overline{r}_2 L_2) \vee (\overline{r}_1 L_3)(\overline{r}_2 L_3)(\overline{r}_3 L_3)$. Its calculation for s_3' is shown in Fig. 6. Note that although two copies of rule $startProcess$ appear in the sequence, they are completed in different ways, thus e.g. $L_3 \overline{r}_1 \neq L_2 \overline{r}_1$.

The following result states conditions to keep coherence in case of permuting one production inside a sequence [11]. We study advancement of the left-most rule to

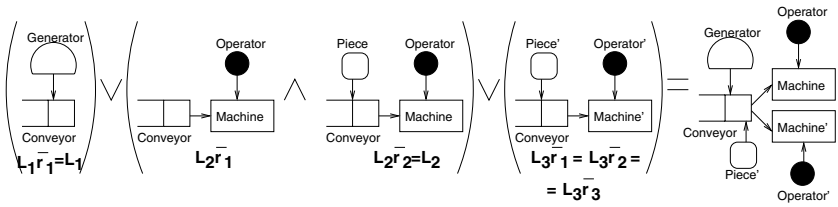


Fig. 6. MID for sequence $s'_3 = startProcess; startProcess; genPiece$.

the front and delay of the right-most rule to the back of a sequence, because these are the most common permutations. However our techniques allow studying other permutations as well.

Theorem 3.4 (Production Permutation) *Let $t_n = p_\alpha; p_n; p_{n-1}; \ldots; p_1$ and $s_n = p_n; p_{n-1}; \ldots; p_1; p_\beta$ be two coherent sequences of productions and let ϕ and δ be two permutations.*

(i) *$\phi(t_n)$ is coherent if:* $e^E_\alpha \bigtriangledown^n_1 \left(\overline{r^E_x}\, L^E_y\right) \vee R^E_\alpha \bigtriangledown^n_1 \left(\overline{e^E_x}\, r^E_y\right) = 0.$

(ii) *$\delta(s_n)$ is coherent if:* $L^E_\beta \bigtriangleup^n_1 \left(\overline{r^E_x}\, e^E_y\right) \vee r^E_\beta \bigtriangleup^n_1 \left(\overline{e^E_x}\, R^E_y\right) = 0.$

where ϕ advances the last production to the front, that is, moves the left-most rule to the right $n-1$ positions in a sequence of n rules. Thus, ϕ is the permutation $\phi = [\,1 \quad n \quad n-1 \ldots 3 \quad 2\,]$. This is a notation for permutation cycles that means that rule 1 (the left-most one) is sent to position n, then rule in position n is moved to position $n-1$, and similarly until rule 3, which is moved to position 2, and this one to position 1. In a similar way, δ delays the first production $n-1$ positions in a sequence of n rules, moving it to the last position. Thus, $\delta = [\,1 \quad 2 \ldots n-1 \quad n\,]$ (i.e. each rule is moved to the right, and rule n to position 1). As an example, for sequence $t_2 = startProcess; repair; breakdown$, $\phi(t_2) = repair; breakdown; startProcess$ is coherent, as we obtain a 0 matrix.

G-congruence guarantees that two compatible concatenations have the same initial digraph G. The conditions to be fulfilled are known as *Congruence Conditions* (CC). The interest of these conditions is that a coherent and compatible concatenation s_n and a coherent and compatible permutation of it, $\sigma(s_n)$, which have the same MID G are *potentially sequential independent*. This means that, when considering a host graph, if the matches of the productions in the sequence coincide with G, then they are sequential independent. Next theorem presents the congruence conditions for advancement and delay of productions (see [11] for the proof):

Theorem 3.5 (G-congruence) *Given sequence s_n, the congruence conditions for rule advance (ϕ_{n-1}) and delay (δ_{n-1}) are given by:*
$CC_n(\phi_{n-1}, s_n) = L_n \nabla^{n-1}_1 (\overline{e_x}\, r_y) \vee r_n \nabla^{n-1}_1 (\overline{r_x}\, L_y) = 0$
$CC_n(\delta_{n-1}, s_n) = L_1 \nabla^n_2 (\overline{e_x}\, r_y) \vee r_1 \nabla^n_2 (\overline{r_x}\, L_y) = 0$

Note that it is possible to check sequential independence between a rule and a sequence, in contrast with results in the categorical approaches. For example, previous sequences t_2 and $\phi(t_2)$ are not G-congruent. The MIDs for t_2 and $\phi(t_2)$

are shown in Fig. 7(a and b). Actually, the congruence condition results in a zero vector, but in a matrix with a 1 in the edges corresponding to the self-loops in the operator and the machine, as well as in the edge from the piece to the machine. These edges are precisely the difference between both MIDs.

On the other hand, sequence $t'_2 = startProcess; genPiece; move$ (where we identify the conveyor of $genPiece$ with the source conveyor of $move$ and the input conveyor of $startProcess$) is G-congruent with $\phi(t'_2) = move; genPiece; startProcess$. This means that they share a common MID (shown in Fig. 7 (c)), and that they output the same result (not the same graph, but an isomorphic one, as the Piece that ends up in $Conveyor'$ is a different one). Note however that we cannot advance $startProcess$ only one step in t'_2. We use symbol $\perp$ for sequential independence, thus writing $startProcess \perp (move; genPiece)$ and $startProcess \not\perp move$ (always relative to the given identification of elements across productions).

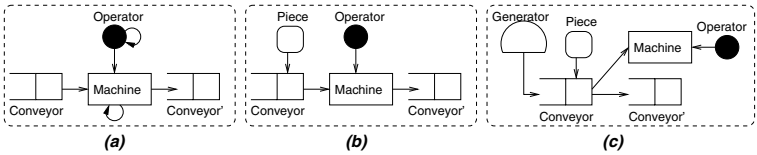


Fig. 7. (a) MID for $startProcess; repair; breakdown$. (b) MID for $repair; breakdown; startProcess$. (c) MID for $startProcess; genPiece; move$ and $move; startProcess; genPiece$.

3.3 Sequence Composition

Next we introduce sequence composition, for which we require the sequence to be compatible. Composition defines a unique production that to a certain extent performs the same actions as the sequence. Recall that compatibility is a means to deal with dangling edges, equivalent to the dangling condition in DPO. When a concatenation of productions is considered, we are not only concerned with the final result but also with intermediate states of the sequence. Compatibility should take this into account and thus a concatenation is said to be compatible if the overall effect on its MID results in a compatible digraph starting from the first production and increasing the sequence until we get the full concatenation. We should then test compatibility for the growing sequence of concatenations $S = \{s_1, s_2, \ldots, s_n\}$ where $s_m = q_m; q_{m-1}; \ldots; q_1$, $1 \leq m \leq n$.

Definition 1 *(Seq. Compatibility) A coherent sequence s_n is* compatible *if the following identity is verified $\forall m \in \{1, \ldots, n\}$:*

$$\left\| \left[s_m\left(M_m^E\right) \vee \left(s_m\left(M_m^E\right)\right)^t \right] \odot \overline{s_m\left(M_m^N\right)} \right\|_1 = 0 \tag{3}$$

where M_m is the minimal initial digraph of sequence s_m.

This definition coincides with the notion of compatibility for one production (see [12]) when the sequence has length one, and with the case of a single graph when considering the identity production.

When we introduced the notion of production, we first defined its LHS and RHS and then we associated some matrices (e and r) to them. The situation for

defining composition is similar, but this time we first observe the overall effect of the production and then decide its left and right hand sides. If $s_n = p_n; \ldots; p_1$ is coherent, then its composition is a production defined by $c = p_n \circ p_{n-1} \circ \ldots \circ p_1$. The description of its erasing and its addition matrices e and r are given by: $S^E = \sum_{i=1}^{n} \left(r_i^E - e_i^E\right)$; $S^N = \sum_{i=1}^{n} \left(r_i^N - e_i^N\right)$. We operate (i.e. perform the composition) through the identified elements across rules in the sequence.

Due to coherence, elements in S^E and S^N are either $+1$, 0 or -1, so they can be split into their positive and negative parts, $S^E = r_+^E - e_-^E$, $S^N = r_+^N - e_-^N$, where all elements in r_+ and e_- are either zero or one. Thus:

Proposition 3.6 *(Composition) Let $s_n = p_n; \ldots; p_1$ be a coherent and compatible concatenation of productions. Then, the composition $c = p_n \circ p_{n-1} \circ \ldots \circ p_1$ defines a production with matrices $r^E = r_+^E$, $r^N = r_+^N$, $e^E = -e_-^E$, $e^N = -e_-^N$, and (L^E, L^N) the MID of s_n.*

The LHS is the minimal digraph necessary to carry out all operations specified by the composition (plus those preserved by the matrix), thus its LHS equals its MID and its RHS is just the image.

Example. Fig. 8 shows the resulting rule of composing *startProcess*; *genPiece*; *move*.

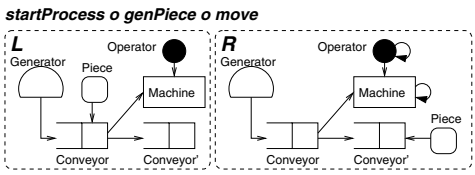


Fig. 8. $startProcess \circ genPiece \circ move$.

Note that the formula for composition coincides with that for the image of the concatenation (see [13]) applied to its MID. This is stated in the next corollary.

Corollary 3.7 *With the above notation, $c(M_n) = s_n(M_n)$.*

4 Derivations

This section introduces the concepts of *match* and *derivation*. Matching is the operation of identifying the LHS of a rule inside a host graph. In our work, we consider only injective matches. Thus, given a production $p : L \to R$ and a simple digraph G, any $m : L \to G$ total injective morphism is a match for p in G. The match can be considerd as one of the ways of *completing* L in G (see section 3.1 and [12]). We do not explicitly care about types in the matching, but this can be thought as restrictions for the *completion* procedure, which cannot identify elements of different types. The following definition of derivation considers not only the elements that should be present in the host graph G, but those that should not, N.

Definition 2 *(Direct Derivation) Given a production $p : L \to R$ as in Fig. 9(a), $d = (p, m)$ – with $m = (m_L, m_N)$ – is called a direct derivation with result $H = p^*(G)$ if the following conditions are fulfilled:*

(i) *There exist* $m_L : L \to G$ *and* $m_N : N \to \overline{G}$ *total injective morphisms, where* $\overline{G} = (\overline{G^E}, G^N)$ *is the negation of graph* G*, constructed by taking the negation of the edge matrix and the nodes vector of* G*.*

(ii) $m_L(n) = m_N(n)$, $\forall n \in L^N \cap N^N$.

(iii) *The square in Fig. 9 commutes* ($m^*_L \circ p\,(L) = p^* \circ m_L\,(L)$) *and is a pushout.*

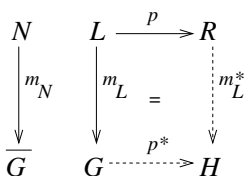


Fig. 9. Direct Derivation

Fig. 10 shows a simple example of derivation, where rule *startProcess* is applied to host graph G. We have also depicted the inclusion of N in $\overline{G}$ (bidirectional arrows have been used for simplification).

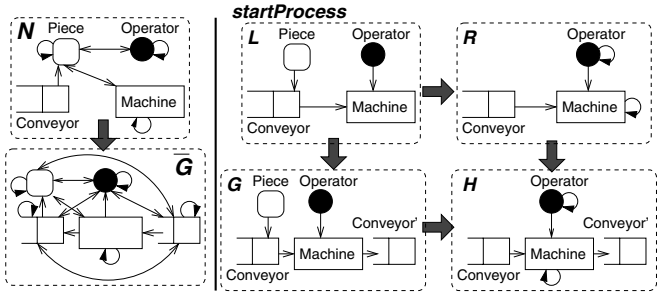


Fig. 10. Direct Derivation Example

When applying a rule to a host graph, *dangling edges* can occur if a connected node in the host graph is deleted by a rule, and the rule does not delete all the connections. This problem is differently addressed in SPO and DPO. In DPO, if an edge becomes dangling then the rule is not applicable for that match, while SPO allows the production to be applied, deleting any dangling edge. In MGGs, we propose an SPO-like behaviour (as DPO can be seen as a special case of SPO). The main idea is that if a rule p produces dangling edges, the rule is enlarged (by means of operator $T_\varepsilon(p)$, see [12]) to explicitly consider the dangling edges in the LHS (by using the extended morphism $m_\varepsilon(L)$, which considers the neighborhood of the original match), and delete them. In [12], we proved that this is equivalent to adding a pre-production (that we call $\varepsilon-$production) to be applied before the original rule (i.e. the original rule p is transformed into sequence $p; p_\varepsilon$). The $\varepsilon-$production deletes the dangling edges and the original rule can be applied as it is. Here we improve that idea, as there is no way to guarantee that when a rule is splitted, both productions are applied to the same elements (in general, matches are non-deterministic). This issue is addressed for example in [14] (for a very different reason) and the solution proposed there is to "pass" the match from one production to the other.

Another possible solution is to define an operator T_μ for a type α acting on production p as follows: if no node has type α in the host graph, then a new node α is added and connected to every already existing node in the RHS of p. If, on the contrary, there exists a node with that type, then it is connected to every node in p's LHS. In essence the idea is to mark nodes with a special type α. Using functional analysis notation: $R = \langle L, p\rangle \mapsto \underline{R} = \langle m_\varepsilon(L), T_\varepsilon(p)\rangle \mapsto \underline{R} = \langle m_\mu \circ m_\varepsilon(L), T_\mu \circ T_\varepsilon(p)\rangle$

Where (as in [12]) $\underline{R}$ is the extended rule's RHS, which considers the dangling edges. Morphism m_μ is quite similar to m_ε in [12] but enlarging L with elements in $dom(m_\varepsilon)\backslash L$.

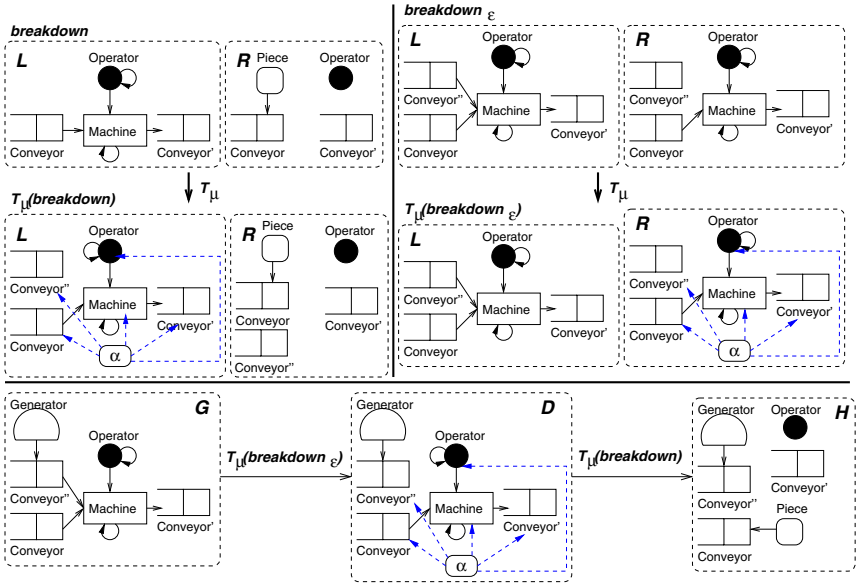


Fig. 11. Marking in sequence $s = breakdown; breakdown_\varepsilon$.

Thus, if p and p_ε are to be applied in the same place, we may proceed as follows: (i) Enlarge p_ε to add one node of some non-existent type (α) together with edges starting in this node and ending in nodes used by p_ε. (ii) Enlarge p to delete the node of type α mentioned in previous point. [3]

Fig.11 illustrates the process for rule *breakdown*. Its application to graph G produces a dangling edge (the one stemming from $Conveyor''$), therefore an ε-production is needed ($breakdown_\varepsilon$) to delete the dangling edge. In addition, operator T_μ is applied to both rules. In case of $breakdown_\varepsilon$, it adds the marking node, for the other rule the operator deletes it. The right of the figure shows the two steps in the derivation. As this process can be easily automated, we can safely ignore it and assume that it is somehow being performed.

Initial Digraph Set. Concerning the MID, the matches in a derivation induce different ways of completing the rule matrices. Thus if we consider them all, we no

[3] Being precise, a new ε-production is created but no recursive process should arise as there shouldn't be any interest in permuting this new ε-production.

longer have a unique MID, but a set. Thus:

Definition 3 *(Initial Digraph Set) Given $s_n = p_n; \ldots; p_1$, the initial digraph set $\mathfrak{M}(s_n)$ is the set of simple digraphs M_i such that $\forall M_i \in \mathfrak{M}(s_n)$ the following properties hold:*

(i) *M_i has enough nodes and edges for every production of the concatenation to be applied in the specified order.*

(ii) *M_i has no proper subgraph with previous property.*

Every element $M_i \in \mathfrak{M}(s_n)$ is an initial digraph for s_n.

The initial digraph set contains all graphs that can potentially be identified by matches in concrete host graphs. The *maximal initial digraph* is the element $M_n \in \mathfrak{M}(s_n)$ that considers all elements in p_i to be different. This graph is unique up to isomorphism, and corresponds to the parallel application of every production in the sequence. In a similar way, graphs $M_i \in \mathfrak{M}(s_n)$ in which all possible identifications are performed are MIDs, which in general are not unique.

5 Applicability

Unless otherwise stated we shall consider s_n to be a sequence of productions and d_n its associated derivation once matchings are found in host graph G. Derivation d_n may contain ε-productions, due to the appearance of dangling edges. We start this section by enunciating the *applicability problem*. Our aim is to characterize applicability with simpler concepts and provide explicit formulae.

Problem 1 (Applicability) *Given sequence s_n (made of rules in grammar $\mathfrak{G}$) and a simple digraph G, is it possible to apply s_n to host graph G?.*

The elements generated by the rules in a sequence that may disturb its applicability are given by one of the parts of the formula for coherence (see Theo. 3.2):

$$\nabla_1^n(\overline{e}_x r_y) = (\overline{e}_1 r_1) \vee (\overline{e}_1 r_2)(\overline{e}_2 r_2) \vee (\overline{e}_1 r_3)(\overline{e}_2 r_3)(\overline{e}_3 r_3) \vee \cdots \qquad (3)$$

This expression can be used to calculate the *negative initial digraph* N for a coherent sequence $s_n = p_n; \ldots; p_1$. It is the smallest simple digraph whose elements cannot be found in the host graph in order to guarantee the applicability of s_n. It is the symmetric concept to that of MID, but for nihilation matrices.

Theorem 5.1 (Negative Initial Digraph) *The minimum digraph that must be found in $\overline{G}$ in order to permit the application of sequence s_n is given by: $N = \nabla_{i=1}^n(\overline{e}_x N_y)$.*

Proof. (Sketch) We can prove the result taking into account elements added by productions in the sequence (but not dangling edges for now) and proceed as in theorem 5.1 in [11]. Then, if necessary, we may use the part of coherence associated to (3) to simplify any cumbersome expression. [4]

[4] It was not used in the demonstration of the minimal initial digraph.

In order to consider not only elements added by previous rules but also dangling edges, it suffices to substitute r_y by N_y, which specifies edges added by rules (r_i) and those incident to nodes which are to be erased (dangling edges). □

Remark. Operations performed by a sequence are generalized by operators ∇ and Δ which represent ascending and descending sequences, e.g., $\nabla_1^3 \overline{e_x} r_y = p_1 p_2(r_3)$ and $\Delta_1^3 \overline{e_x} r_y = p_3 p_2(r_1)$. *Generalization* in the sense that it allows the application of this operational structure but not limited to matrices e and r, e.g. $\bigtriangledown_2^5 \overline{e}_x L_x \, (\overline{r}_y \vee L_y)$.

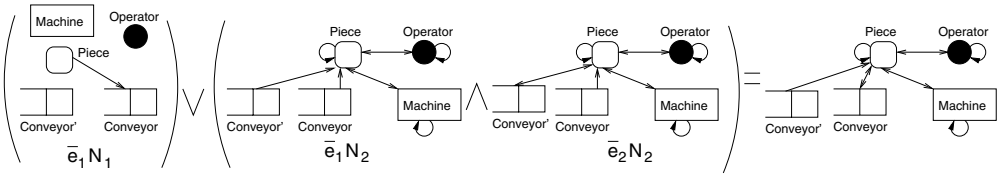


Fig. 12. Negative Initial Digraph for sequence $startProcess; move$.

Fig. 12 shows the negative initial digraph for sequence $startProcess; move$, where the target conveyor in *move* is identified with the input one in $startProcess$. The resulting graph shows that the piece cannot have any connection, except the one explicitly removed by rule *move*, as the *Piece* is deleted by the second rule, and otherwise would produce dangling edges. Moreover, neither the *Operator* nor the *Machine* can have self-loops. The example shows very clearly the need to complete matrices of all graphs before proceeding to the calculations, as otherwise $\overline{e}_1 N_1$ and $\overline{e}_1 N_2$ would not take into account the edges from *Piece* to *Conveyor′*.

Asking for coherence and compatibility (refer to [11]) of d_n is equivalent to finding its minimal and negative initial digraphs in the host graph and its negation, respectively. Applicability can be fully characterized in terms of coherence and compatibility or minimal and negative initial digraphs.

Theorem 5.2 (Characterization) *Sequence s_n is applicable with respect to G if there are matches for every production such that:*

- *derivation d_n is coherent and compatible*

or, equivalently,

- *its minimal initial digraph is in G and its negative initial digraph is in $\overline{G}$.*

Proof. (Sketch) If productions are well defined (in the sense of definition 3.2 in [11]) then compatibility is guaranteed by ε-productions.

Coherence depends on the node identification performed by matches (the so-called *actual initial digraph* in [12]) and its formula is equivalent in some sense (or guaranteed) if some actual initial digraph and negative initial digraphs (precisely those given by identifications proposed by matches) are respectively found in G and $\overline{G}$ (see definition 2). □

Next, we enunciate the reachability problem, which is an extension of applicability as introduced in problem 1.

Problem 2 (Reachability) *For two given states (initial S_0 and final S_T), is there any sequence that transforms S_0 into S_T?.*

For Petri nets there is an algebraic characterization deriving the so called *state equation*, which we generalized to cope with Matrix Graph Grammars in [13].

6 (Sequential) Independence

Sequential independence for derivations can be stated similarly to problems 1 and 2. Here σ will represent an element of the group of permutations and derivation d_n will have associated sequence s_n. Note that two sequences s_n and $s'_n = \sigma(s_n)$ carry out the same operations but in different order.

Problem 3 (Independence) *For two given derivations d_n and d'_n applicable to host graph G, do they reach the same state?.*

Note the close similarity with *local confluence* [4]. The problem can be easily extended to consider any finite number of derivations. Again, our objective is to characterize under which circumstances, depending on the permutation applied and on the definition of the grammar (which includes both grammar rules and the host graph), it is possible to conclude that their final states are isomorphic.

Problem 4 (Sequential Independence) *For two given derivations d_n and $d'_n = \sigma(d_n)$ applicable to host graph G, do they reach the same state?.*

In both cases there is a dependence relationship w.r.t. problem 1. Problem 2 is also related to problems 3 and 4: every solution of the state equation specifies the productions to be applied but not the order. Sequences associated to different solutions of the state equation are independent but may not be sequential independent. Thus, reachability *splits* independence and sequential independence.

Definition 4 *(Sequential Independence) Two derivations d_n and $d'_n = \sigma(d_n)$ are sequential independent w.r.t. G if $d_n(G) = H_n \cong H'_n = d'_n(G)$.*

Note that even though $s'_n = \sigma(s_n)$, if ε-productions appear then it may not be true that $d'_n = \sigma(d_n)$, unless they are equal. A restatement of def. 4 is the following proposition. The existence of an initial digraph guarantees coherence for both derivations.

Proposition 6.1 *If for two applicable derivations d_n and $d'_n = \sigma(d_n)$ $\exists M_0 \subset G$ such that $\emptyset \neq M_0 \in \mathfrak{M}(s_n) \cap \mathfrak{M}(s'_n)$ then $d_n(M_0)$ and $d'_n(M_0)$ are sequential independent.*

Proof. Apply results in [11], composition in particular. □

In order to calculate M_0 in prop. 6.1 it is possible to follow two complementary approaches: either we start by the *maximal initial digraph* or by different minimal elements in the initial digraph set. In the first case the following identity may be of some help:

$$M_{d_n} = M_{d'_n} \Leftrightarrow M_{d_n} M_{d'_n} \vee \overline{M}_{d_n} \overline{M}_{d'_n} = 1 \tag{4}$$

For the maximal initial digraph M, every element is different across productions in derivations. Let all elements (except those already known) be represented by variables in M and use a SAT solver on (4) to obtain conditions. The same can be applied to the negative intial digraph to guarantee applicability.

If two derivations (with underlying permuted sequences) are not a permutation of each other due to ε-productions but are confluent, then in fact it is possible to write them as a permutation of each other:

Proposition 6.2 *If d_n and d'_n are sequential independent and $s'_n = \sigma(s_n)$, then $\exists\, \hat{\sigma} \mid d'_n = \hat{\sigma}(d_n)$ for some appropriate composition of ε-productions.*

Proof. Let $\hat{T}_\varepsilon : p_\varepsilon \mapsto \hat{T}(p_\varepsilon)$ be an operator acting on ε-productions, which splits them into a sequence of n productions, each one of them deleting one edge. If $\hat{T}_\varepsilon$ is applied to d_n and d'_n we must get the same number of ε-productions. Morover, the number must be the same for every type of edge or a contradiction can be derived as ε-productions only delete elements. □

Example. Assume we have rules *release* and *remove* and a host graph G as shown in Fig. 13. Suppose we want to apply sequences $s_2 = remove; release$ and $s'_2 = release; remove$, identifying the released machine and the one to be removed. With this identification *remove* and *release* are not sequential independent. If we apply s'_2, then an ε-production (deleting the edge from the operator to the machine) has to be added to the derivation, leading to $d'_2 = release; remove; remove_\varepsilon$ which makes *release* inapplicable. However, if in both sequences we identify separately the released and the removed machine, then both sequences are applicable obtaining graph H, and thus $remove \perp release$ for this particular identification. Note that M_0 is the actual initial digraph for this identification and that $M_0 \in \mathfrak{M}(s_2) \cap \mathfrak{M}(s'_2)$ (see Fig. 13). This agrees with previous propositions because there is sequential independence when *remove* does not generate any associated ε-productions. Note how, the explicit deletion of dangling edges by means of productions facilitates the study of rule independence.

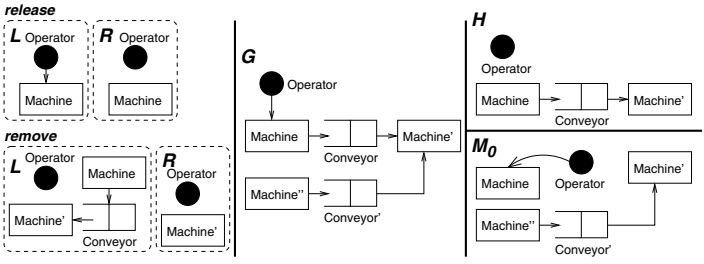


Fig. 13. Sequential independence with *free* matching.

The theory we developed in [11] (without considering the host graph) fits very well here and all results for sequential independence are recovered. Moreover, we can relate the corresponding theorems in [11] for advancement and delaying of productions with composition. One interesting point is that we can study a priori the conditions that need to be fulfilled in order to obtain sequential independence and interpret them as graph constraints or application conditions.

7 Parallel Independence

In this section we analize which productions or group of productions can be computed in parallel and what conditions guarantee this operation, for the moment without considering the host graph (or assuming that no $\varepsilon-$productions are produced by the derivation).

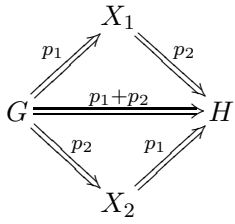


Fig. 14. Parallel Execution.

In the categorical approach the definition for two productions is settled considering the two alternative sequential ways in which they can be composed, looking for sameness in their final state. Intermediate states are disregarded using categorical coproduct of the involved productions. Then, the main difference between sequential and parallel execution is the existence of intermediate states in the former, as seen in Fig. 14. We follow the same approach saying that it is possible to execute two productions in parallel if the result does not depend on generated intermediate states. However, in DPO, it is possible to identify different elements in the parallel rule $(p_1 + p_2)$ to the same element in the host graph through non-injective matches. In our case we have to decide which elements will get identified before performing the composition.

Definition 5 *(Parallel Independence) Productions p_1 and p_2 are truly concurrent if it is possible to define their composition and it does not depend on the order: $p_2 \circ p_1 = p_1 \circ p_2$.*

We use the notation $p_1 \parallel p_2$ to denote true concurrency (i.e. parallel independence). It defines a symmetric relation so it does not matter whether $p_1 \parallel p_2$ or $p_2 \parallel p_1$ is written.

Next proposition compares parallel and sequential independence for two productions, in the style of the *parallelism theorem* (see [2]). The proof is straightforward in our case and is not included.

Proposition 7.1 *Let $s_2 = p_2; p_1$ be a coherent concatenation and assume compatibility, then: $p_1 \parallel p_2 \iff p_2 \perp p_1$.*

Proof. Assuming compatibility frees us from ε-productions. Elements are identified in the same way in $p_1 \parallel p_2$ and $p_2 \perp p_1$. □

So far we have just considered one production per branch when parallelizing, as represented in Fig. 14. One way to deal with more general schemes (see Fig. 15) is to test parallelism for each element in one branch against every element in the

other. In the figure, sequences $s_1 = p_6; p_5; p_4$ and $s_2 = p_3; p_2; p_1$ can be computed in parallel if there is sequential independence for every interleaving. This is true if $p_i \parallel p_j$, $\forall i \in \{4,5,6\}$, $\forall j \in \{1,2,3\}$. There are many combinations that keep the relative order of s_1 and s_2, e.g. $p_6; p_3; p_2; p_5; p_1; p_4$ or $p_3; p_6; p_2; p_5; p_1; p_4$. In order to apply these two sequences in parallel, all interleavings that maintain the relative order should have the same result.

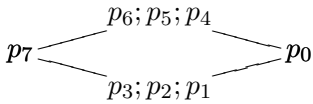


Fig. 15. Parallel Execution Example.

Though there are some similarities between this concept and the *concurrency theorem* [4], here we rely on the possibility to characterize production advancement or delaying inside sequences more than just one position, hence, being more general.

Theorem 7.2 *Let $s_n = p_n; \ldots; p_1$ and $t_m = q_m; \ldots; q_1$ be two compatible and coherent sequences with the same MID, where either $n = 1$ or $m = 1$. Suppose $r_{m+n} = t_m; s_n$ is compatible and coherent and either $t_m \perp s_n$ or $s_n \perp t_m$. Then, $t_m \parallel s_n$ through composition.*

Proof. Using proposition (7.1). □

Through composition means that the concatenation with length greater than one must be transformed into a single production using composition. This is possible because it is coherent and compatible (see prop. 3.6). In fact it would not be necessary to transform the whole concatenation using composition, but only the parts that present a problem.

Setting $n = 1$ corresponds to advancing a production in sequential independence, while $m = 1$ to moving a production backwards inside a concatenation. In addition, in the hypothesis we ask for coherence of r_n and either $t_m \perp s_n$ or $s_m \perp t_n$. In fact, if r_{m+n} is coherent and $t_m \perp s_n$, then $s_n \perp t_m$. It is also true that if r_{m+n} is coherent and $s_n \perp t_m$, then $t_m \perp s_n$ (it could be proved by contradiction).

The idea behind Theo. 7.2 is to erase intermediate states through composition but, in a real system, this is not always possible or desirable if for example these states were used for synchronization of productions or states.

8 Related Work

The literature for SPO and DPO has mainly studied pair of rules, whereas in our approach we consider derivations of arbitrary finite length. The only study for derivations we are aware of is *shift-equivalence* [2], which is a relation between derivations (used in models of computation for graph grammars) that equates them if they are related by a finite number of one-step advancements of a rule inside one of the derivations. This is modelled through the application of analysis and synthesis

operations on parallel rules (made of a set of rules where each rule is parallel-independent of all the others) and derivations respectively. In addition, in order to compute shift-equivalent *canonical derivations* [2] (where each rule is applied as soon as possible), rules are advanced in single steps, but only if they are independent with the following one. As we have seen in the example for G-congruence, our notion of independence for derivations is stronger, as a rule can be advanced two (or more) positions even if it is not independent with the following one. Moreover, we explicitly give the conditions, instead of first assuming independence and then using categorical operators. In addition, we believe that explicitly modelling the deletion of dangling edges by means of ε-productions facilitates this study.

With respect to composition, note that we identify accross rules the elements through which composition is performed. This is similar to the concurrency theorem [4]. Non-injective matches in DPO allow identifying different elements in a parallel rule, while we have to decide such identification before calculating the composition.

With respect to other similar approaches to Matrix Graph Grammars, in [15] the DPO approach was implemented using Mathematica. In that work, (simple) digraphs were represented with their boolean adjacency matrices. This is the only similarity with our work, as our goal is to develop a theory for (simple) graph rewriting based on boolean matrix algebra. Other somehow related work is the relational approaches of [7] and [10]. However, they rely on category theory for expressing the rewriting.

9 Conclusions and Future Work

In this paper we have presented some new concepts of MGGs. In particular we have introduced the nihilation matrix and the negative initial digraph, which make explicit the elements that must not be present in a rule or sequence for their application. We have also studied applicability of sequences and sequential independence (for derivations). This latter concept has been extended to sequences of arbitrary finite length. Our approach of explicitly modelling the deletion of dangling edges by means of ε-productions greatly facilitates this analysis. We have also introduced rule composition and parallel independence in the absence of dangling edges.

The next step after problem 4 is the study of confluence [4,6], which can be settled as a problem very much like those introduced so far. We are also working on the study of parallel independence in the presence of $\varepsilon-$productions, application conditions and tool support, taking AToM^3 [8] as a basis.

Acknowledgement

Work sponsored by Spanish Ministry of Science and Education, project MOSAIC (TSI2005-08225-C07-06).

References

[1] Braket notation intro: http://en.wikipedia.org/wiki/Bra-ket_notation

[2] Corradini, A., Montanari, U., Rossi, F., Ehrig, H., Heckel, R., Löwe, M. 1999. *Algebraic Approaches to Graph Transformation - Part I: Basic Concepts and Double Pushout Approach*. In [5], pp.: 163-246

[3] Ehrig, H., Heckel, R., Korff, M., Löwe, M., Ribeiro, L., Wagner, A., Corradini, A. 1999. *Algebraic Approaches to Graph Transformation - Part II: Single Pushout Approach and Comparison with Double Pushout Approach*. In [5], pp.: 247-312.

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

[5] Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. 1999. *Handbook of Graph Grammars and Computing by Graph Transformation. Vol 1.* World Scientific.

[6] Huet, G., Oppen, D. C. 1980. *Equations and Rewrite Rules: A Survey*. Tech. Rep. STAN-CS-80-785 Computer Science Department, Stanford University.

[7] Kahl, W. 2002. *A Relational Algebraic Approach to Graph Structure Transformation*. Tech. Rep. 2002-03, Universitat der Bundeswehr Munchen.

[8] de Lara, J., Vangheluwe, H. 2004. *Defining Visual Notations and Their Manipulation Through Meta-Modelling and Graph Transformation*. JVLC, Vol 15(3-4), pp.: 309-330. Elsevier.

[9] Mens, T., Taentzer, G., Runge, O. 2007. *Analysing refactoring dependencies using graph transformation* To appear in Software and Systems Modeling. Springer.

[10] Mizoguchi, Y., Kuwahara, Y. 1995. *Relational Graph Rewritings*. TCS 141:311–328, Elsevier.

[11] Pérez Velasco, P. P., de Lara, J. 2006. *Towards a New Algebraic Approach to Graph Transformation: Long Version*. Tech. Rep. of the School of Comp. Sci., Univ. Autónoma Madrid. http://www.ii.uam.es/~jlara/investigacion/techrep_03_06.pdf.

[12] Pérez Velasco, P. P., de Lara, J. 2006. *Matrix Approach to Graph Transformation: Matching and Sequences*. Proc. ICGT'06, LNCS 4218, pp.:122-137. Springer.

[13] Pérez Velasco, P. P., de Lara, J. 2006. *Petri Nets and Matrix Graph Grammars: Reachability*. Proc. PN-GT'06, Electronic Communications of EASST(2).

[14] Schürr, A. 1994. *Specification of Graph Translators with Triple Graph Grammars*. LNCS 903, pp.: 151 - 163. Springer.

[15] Valiente, G. 1998. *Grammatica: An Implementation of Algebraic Graph Transformation on Mathematica*. Proc. 6^{th} Works. on Theory and Application of Graph Transformations. pp. 261–267.