Processes for Adhesive Rewriting Systems*

Paolo Baldan[1], Andrea Corradini[2], Tobias Heindel[3], Barbara König[3], and Paweł Sobociński[4]

[1] Dipartimento di Informatica, Università Ca' Foscari di Venezia, Italy
[2] Dipartimento di Informatica, Università di Pisa, Italy
[3] Institut für Formale Methoden der Informatik, Universität Stuttgart, Germany
[4] Computer Laboratory, University of Cambridge, United Kingdom

Abstract. Rewriting systems over adhesive categories have been recently introduced as a general framework which encompasses several rewriting-based computational formalisms, including various modelling frameworks for concurrent and distributed systems. Here we begin the development of a truly concurrent semantics for adhesive rewriting systems by defining the fundamental notion of process, well-known from Petri nets and graph grammars. The main result of the paper shows that processes capture the notion of true concurrency—there is a one-to-one correspondence between concurrent derivations, where the sequential order of independent steps is immaterial, and (isomorphism classes of) processes. We see this contribution as a step towards a general theory of true concurrency which specialises to the various concrete constructions found in the literature.

1 Introduction

Many rewriting theories have been developed in order to describe rule-based transformations over specific classes of objects: words (formal languages), terms, multi-sets (Petri nets) and graphs (graph rewriting). The recently introduced categorical foundation for double-pushout (DPO) rewriting theory based on *adhesive categories* [13] encompasses rewriting on words, multi-sets and (typed) graphs. Indeed, adhesive categories satisfy practically all the High-Level Replacement conditions [8], which ensure the validity of several standard theorems.

As a consequence of the relatively simple axioms and closure properties of adhesive categories, it is not difficult to show that a wide range of structures form the objects of an adhesive category. For instance, the categories of graphs with second-order edges or graphs with scopes are adhesive. Because of their generality, adhesivity and related concepts have begun to be exploited in the area of graph transformation (see e.g., [9]).

The view of adhesive rewriting systems as a general, unifying setting into which several models of concurrent and distributed systems can be embedded, calls for a generalization of the concurrency theory already developed for specific

* Partially supported by EPSRC grant GR/T22049/01, DFG project SANDS, EC RTN 2-2001-00346 SegraVis, MIUR project PRIN 2005015824 ART and EU IST-2004-16004 SEnSOria.

L. Aceto and A. Ingólfsdóttir (Eds.): FOSSACS 2006, LNCS 3921, pp. 202–216, 2006.

formalisms like Petri nets and graph rewriting to this framework. The first steps in this direction were already taken in [13] where the notions of sequential and parallel independence of two rewriting steps, i.e. conditions under which they can be switched or applied concurrently, were studied.

In this paper we continue the development of a truly concurrent semantics for adhesive rewriting systems by generalizing the fundamental notion of process, well-known from the theory of Petri nets [11]. A process describes a possible computation of a given rule-based system taking into account the dependencies between the rewriting steps. The fact that two events are concurrent is modeled by the absence of dependencies between them. Intuitively, a process provides a canonical representation of a class of *derivations* (sequences of rewriting steps) which differ only in the order of independent rewriting steps.

The theory of processes and their correspondence with suitable equivalence classes of derivations has been generalized from nets to graph transformation systems in [6, 4, 1]. These approaches rely on the set-theoretical concept of *items*—tokens in the case of Petri nets, nodes and edges in the case of graph rewriting. For example, a transition t is said to be a *cause* of another transition t' if it produces a token in the pre-set of t', while in DPO-graph rewriting a rule cannot be applied to a graph if it deletes a node without deleting all edges incident to it (the so-called *dangling condition*).

In the abstract setting of adhesive categories, a concept related to the notion of *item* is that of *subobject* of an object X. A subobject is an isomorphism class of monomorphisms into X. For example, in the category of sets and functions, the subobjects of a set are (in 1-1-correspondence with) its subsets, while in the category of graphs and homomorphisms, a subobject of a graph is a subgraph. When working with subobjects of an object X in adhesive categories, we benefit from the fact that they form a distributive lattice [13]. However, we have no notion of "atoms" that can be consumed or produced. As a consequence, the techniques involved in the development of our theory are significantly different from those used in the setting of nets or graph rewriting and an original approach is needed in order to deal with the relevant concepts such as causality, concurrency, and negative application conditions for rules.

From a theoretical perspective, the central merit of our development lies in readdressing in the abstract setting of adhesive categories the concept of process that has so far been defined only in concrete cases. This is in contrast to related notions such as parallel and sequential-independence which are traditionally defined at the abstract level. The advantages of understanding processes at a general level are clear: we are able to prove theorems without resorting to the use of low-level structure.

The theory in this paper provides the foundations for the development of partial order verification methods that are applicable to rewriting systems over general "graph-like" structures, including, for instance, UML models, bigraphs and dynamic heap-allocated pointer structures.

Structure of the paper. We recall the definition of adhesive categories as well as some of their properties in §2. Adhesive grammars and derivations are introduced

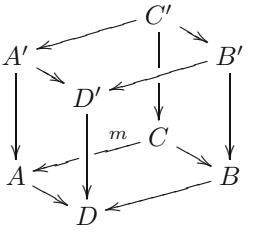
in §3 followed by a study of the possible relations among the rules and their connections with concurrency. The notion of occurrence grammars (on which the notion of process is based) is developed in §4. Finally, in §5 we define processes and show that processes and switch-equivalence classes of typed derivations are in 1-1-correspondence.

2 Adhesive Categories

Adhesive categories were introduced in [13]. Roughly, they may be described as categories where pushouts along monomorphisms are "well behaved". Here we only give a minimal introduction, concentrating on the algebra of subobjects of a given object T.

Definition 1 (Adhesive category). A category $\mathbf{C}$ is said to be *adhesive* if

1. $\mathbf{C}$ has pushouts along monomorphisms;
2. $\mathbf{C}$ has pullbacks;
3. Given a cube diagram as shown to the right with: (*i*) $m\colon C \to A$ mono, (*ii*) the bottom face a pushout and (*iii*) the back faces pullbacks, we have that the top face is a pushout iff the front faces are pullbacks.



The archetypal adhesive category is the category **Set** of sets and functions. Adhesive categories enjoy useful closure properties, for example if $\mathbf{C}$ is adhesive then so is any functor category $\mathbf{C}^{\mathbf{X}}$, any slice category $\mathbf{C} \downarrow C$ and any co-slice category $C \downarrow \mathbf{C}$. Therefore, since the category of graphs and graph morphisms is a functor category $\mathbf{Graph} \cong \mathbf{Set}^{\bullet \rightrightarrows \bullet}$, it is adhesive, and given a type graph T, the category of typed graphs $\mathbf{Graph} \downarrow T$ is adhesive.

A subobject of a given object T is an isomorphism class of monomorphisms to T. Binary intersections of subobjects exist in any category with pullbacks. Adhesive categories enjoy also the existence of binary subobject unions which are calculated in an intuitive way by pushing out along their intersection. Moreover, the lattice of subobjects is distributive.

Theorem 2 ([13], Theorem 17 and Corollary 18). *For an object T of an adhesive category $\mathbf{C}$, the poset* $\mathrm{Sub}(T)$ *of subobjects of T has joins: the join of two subobjects is (the isomorphism class of) their pushout in $\mathbf{C}$ over their intersection. Furthermore the lattice* $\mathrm{Sub}(T)$ *is distributive.*

3 Adhesive Grammars

We start by introducing rules and grammars. Rules consist of three objects: a left-hand side, a right-hand side and a common "read-only" part that is preserved, called the interface, which is a subobject of both the left- and the right-hand side.

Definition 3 (Rules and grammars). Let $\mathbf{C}$ be an adhesive category that we assume to be fixed for the rest of the paper. A *rule* is a span of monomorphisms $L \xleftarrow{\alpha} K \xrightarrow{\beta} R$ in $\mathbf{C}$. It is called *consuming* if α is not an isomorphism.

A *grammar* is a triple $\mathcal{G} = \langle S, P, \pi \rangle$, where P is a set of *rule names*, π is a function which maps any $q \in P$ to a rule $L_q \xleftarrow{\alpha_q} K_q \xrightarrow{\beta_q} R_q$ and $S \in \mathrm{ob}(\mathbf{C})$ is the *start object*. The grammar $\mathcal{G}$ is called consuming if all its rules are consuming.

A direct derivation is a diagram representing a single application of a rewriting rule. Applying several rules in sequence gives us a path through the state space of the grammar. The diagram consisting of the corresponding sequence of direct derivations can be reconstructed from a given path, and together they form a derivation.

Definition 4 (Direct derivations and paths). Let $\mathcal{G} = \langle S, P, \pi \rangle$ be a grammar, let $q \in P$, $A, B \in \mathrm{ob}(\mathbf{C})$, and $f \colon L_q \rightarrowtail A$ be a monomorphism. Then q *rewrites A to B at f in $\mathcal{G}$*, written $A \overset{\langle q,f \rangle}{\Longrightarrow}_{\mathcal{G}} B$, if there exists a diagram (1) consisting of two pushouts. If it exists, we shall refer to such a diagram as a *direct derivation along* $\langle q, f \rangle$, to D as *pushout complement* of α_q and f, and to f as a *(q-)match*.

$$\begin{array}{ccccc} L_q & \xleftarrow{\alpha_q} & K_q & \xrightarrow{\beta_q} & R_q \\ {\scriptstyle f}\downarrow & \ulcorner & \downarrow{\scriptstyle g} & \urcorner & \downarrow{\scriptstyle h} \\ A & \xleftarrow[\gamma]{} & D & \xrightarrow[\delta]{} & B \end{array} \tag{1}$$

A $\mathcal{G}$*-path* is a sequence $\tau = \langle q_i, f_i \rangle_{i \in [n]}$, so that $A_0 = S$ and $A_i \overset{\langle q_i, f_i \rangle}{\Longrightarrow}_{\mathcal{G}} A_{i+1}$ for $i \in [n]$.[1] Given a $\mathcal{G}$-path τ, let $\boldsymbol{d}^\tau$ be the diagram which results from including the direct derivations of all of τ's individual steps:

$$\begin{array}{ccccccccccccc} & & L_0 & \xleftarrow{\alpha_0} & K_0 & \xrightarrow{\beta_0} & R_0 & & L_1 & \xleftarrow{\alpha_1} & K_1 & \xrightarrow{\beta_1} & R_1 & \cdots & & & L_{n-1} & \xleftarrow{\alpha_{n-1}} & K_{n-1} & \xrightarrow{\beta_{n-1}} & R_{n-1} \\ & {\scriptstyle f_0}\swarrow & & \downarrow{\scriptstyle g_0} & & \searrow{\scriptstyle h_0} & & {\scriptstyle f_1}\swarrow & & \downarrow{\scriptstyle g_1} & & \searrow{\scriptstyle h_1} & & \ldots & {\scriptstyle f_{n-1}}\swarrow & & \downarrow{\scriptstyle g_{n-1}} & & \searrow{\scriptstyle h_{n-1}} \\ S = A_0 & & \xleftarrow[\gamma_0]{} & D_0 & \xrightarrow[\delta_0]{} & & A_1 & \xleftarrow[\gamma_1]{} & D_1 & \xrightarrow[\delta_1]{} & & A_2 & \cdots & A_{n-1} & \xleftarrow[\gamma_{n-1}]{} & D_{n-1} & \xrightarrow[\delta_{n-1}]{} & & A_n \\ & & \underbrace{\qquad\qquad}_{\boldsymbol{d}_0^\tau} & & & & & \underbrace{\qquad\qquad}_{\boldsymbol{d}_1^\tau} & & & & & & & & \underbrace{\qquad\qquad}_{\boldsymbol{d}_{n-1}^\tau} & & & & \end{array}$$

Then $\boldsymbol{d}^\tau$ is said to be a *diagram of* τ and a *witness* of $A_0 \overset{\tau}{\Rightarrow} A_n$ and the pair $\langle \tau, \boldsymbol{d}^\tau \rangle$ is called a $\mathcal{G}$*-derivation*. For each $i \in [n]$ we write $\boldsymbol{d}_i^\tau$ for the sub-diagram of $\boldsymbol{d}^\tau$ that witnesses $A_i \overset{\langle q_i, f_i \rangle}{\Longrightarrow} A_{i+1}$, and $\boldsymbol{d}_{[i]}^\tau$ for the sub-diagram containing the first i steps of the derivation diagram. Each sub-diagram $L_i \xleftarrow{\alpha_i} K_i \xrightarrow{\beta_i} R_i$ is said to be an *occurrence of* q_i.

In the sequel we will consider typed grammars, as introduced in [6], which are grammars where every component is endowed with a morphism into a fixed

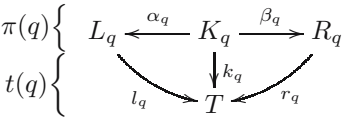
[1] For each $n \in \mathbb{N}$, we denote by $[n]$ the set $\{0, \ldots, n-1\}$.

object $T \in \mathrm{ob}(\mathbf{C})$. Roughly, the type object T is intended to provide the pattern which any possible system state must conform to, and the existence of the typing morphism $a\colon A \to T$ ensures that the state A conforms to the type.

Formally, typed grammars can be seen as grammars in the slice category $\mathbf{C}{\downarrow}T$, which is adhesive when $\mathbf{C}$ is (see [13]). However having an explicit typing will be useful when defining the process of a grammar $\mathcal{G}$, which describes a concurrent computation in $\mathcal{G}$ by representing the rules and the resources used in such a computation. Explicitly working with this type object will enable us to view all left-hand sides, right-hand sides and interfaces as subobjects and work in the subobject lattice $\mathrm{Sub}(T)$.

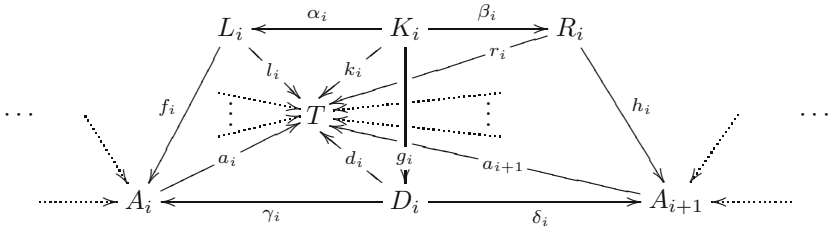
To describe the typed setting formally it shall be convenient to consider an "identity" rule for the start object of a grammar. Given $S \in \mathrm{ob}(\mathbf{C})$, we shall adopt the convention of letting $\underline{S}$ denote the rule $\pi(\underline{S}) = S \xleftarrow{\mathrm{id}} S \xrightarrow{\mathrm{id}} S$.

Definition 5 (Typed grammars and typed derivations). A *typed* grammar is a tuple $\mathcal{G} = \langle \mathcal{G}', T, t\rangle$ where $\mathcal{G}' = \langle S, P, \pi\rangle$ is a grammar, $T \in \mathrm{ob}(\mathbf{C})$ is the *type object* and t is the *(rule) typing*, which assigns to each rule name $q \in P \cup \{\underline{S}\}$ a cocone (span in $\mathbf{C} \downarrow T$) for $\pi(q)$ to T as depicted in the commutative diagram below.



A rule q is called *mono-typed* if l_q and r_q are monos; $\mathcal{G}$ is called *mono-typed* if all $q \in P \cup \{\underline{S}\}$ are mono-typed.

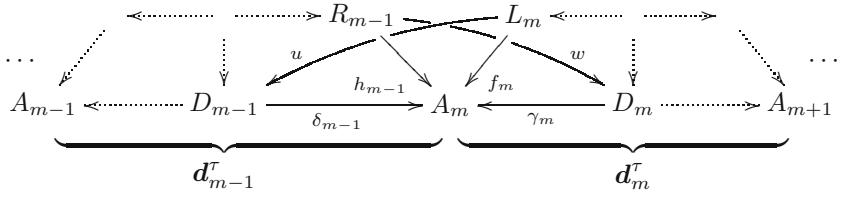
Let $\mathcal{G} = \langle \mathcal{G}', T, t\rangle$ be a typed grammar, where $\mathcal{G}' = \langle S, P, \pi\rangle$; then a *typed $\mathcal{G}$-derivation* is a triple $\rho = \langle \tau, \boldsymbol{d}^\tau, c\rangle$ where $\langle \tau, \boldsymbol{d}^\tau\rangle$ is a $\mathcal{G}'$-derivation and c is a cocone to T for $\boldsymbol{d}^\tau$ that coincides with $t(q)$ on each rule occurrence of q in $\boldsymbol{d}^\tau$ for each $q \in P \cup \{\underline{S}\}$.



The grammar $\mathcal{G}$ is called *safe* if all objects reachable from the start object are mono-typed.

Consider two rules q_{m-1}, q_m which can be applied in sequence and rewrite A_{m-1} to A_m and then to A_{m+1}, as shown in the next diagram. Furthermore assume that the left-hand side of q_m is already present in D_{m-1} and the right-hand side of q_{m-1} can still be found in D_m. This means that these rules do not interfere with each other and their applications can hence be switched, leading to the same result A_{m+1}. Pairs of direct derivations of this kind are called sequential-independent.

Definition 6 (Sequential independence [7]). Let $\langle \tau, \boldsymbol{d}^\tau \rangle$ be a derivation. Then, fixing $m \in \big[\!\big[|\tau|\big]\!\big]$, $m > 0$, the direct derivations $\boldsymbol{d}^\tau_{m-1}$ and $\boldsymbol{d}^\tau_m$ are *sequential-independent* if there are morphisms $u\colon L_m \to D_{m-1}$ and $w\colon R_{m-1} \to D_m$ such that the diagram below commutes, i.e., $\delta_{m-1} \circ u = f_m$ and $\gamma_m \circ w = h_{m-1}$.



We shall now introduce certain relations between the rules of a mono-typed grammar, and the resulting connections with sequential independence and the classical Local Church-Rosser Theorem. In the following, the inclusion (or partial order) $\sqsubseteq$, union (or join) $\sqcup$ and intersection (or meet) $\sqcap$ are interpreted in the subobject lattice $\mathrm{Sub}(T)$.

Definition 7 (Rule relations). Let $\mathcal{G} = \big\langle \langle S, P, \pi \rangle, T, t \big\rangle$ be a mono-typed grammar and let $q, q' \in P$ be rule names. We define four *rule relations*:

$<$ **:** q *directly causes* q', written $q < q'$, if $R_q \sqcap L_{q'} \not\sqsubseteq K_q$
$\ll$ **:** q *can be disabled by* q', written $q \ll q'$, if $L_q \sqcap L_{q'} \not\sqsubseteq K_{q'}$
$<_{\mathsf{co}}$ **:** q *directly co-causes* q', written $q <_{\mathsf{co}} q'$, if $R_q \sqcap L_{q'} \not\sqsubseteq K_{q'}$
$\ll_{\mathsf{co}}$ **:** q *can be co-disabled by* q', written $q \ll_{\mathsf{co}} q'$, if $R_q \sqcap R_{q'} \not\sqsubseteq K_q$.

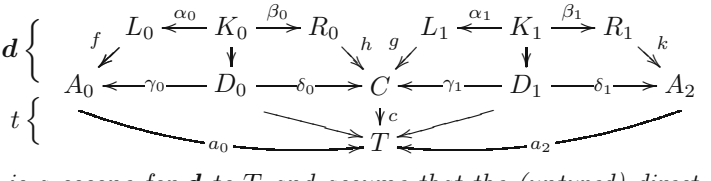
The following proposition gives a partial account of the relationship between sequential independence and rule relations.

Proposition 8. *Let $\langle \tau, \boldsymbol{d}^\tau, c \rangle$ be a typed derivation such that $\boldsymbol{d}^\tau$ witnesses $A_0 \overset{\langle q_0, f \rangle}{\Longrightarrow} C \overset{\langle q_1, g \rangle}{\Longrightarrow} A_2$ and suppose that C is mono-typed. Then:*

1. *If $q_0 \not< q_1$ and $q_0 \not\ll q_1$ then $\boldsymbol{d}^\tau_0$ and $\boldsymbol{d}^\tau_1$ are sequential-independent;*
2. *If $\boldsymbol{d}^\tau_0$ and $\boldsymbol{d}^\tau_1$ are sequential-independent then $q_0 \not< q_1$ and $q_0 \not<_{\mathsf{co}} q_1$.*

As mentioned above, sequential-independent direct derivations can be switched, giving us the first part of the following result. Moreover, when working with mono-typed grammars and derivations, we identify a sufficient condition making it possible to construct the "middle-object" of the switched derivation as a subobject of the type object.

Theorem 9 (Local Church-Rosser). *Consider the typed derivation diagram below:*



where t is a cocone for $\boldsymbol{d}$ to T and assume that the (untyped) direct derivations are sequential-independent. Then the following hold:

1. *There exist* C', g', f' *and a witness* $\boldsymbol{d}'$ *for* $A_0 \xrightarrow{\langle q_1, g' \rangle} C' \xrightarrow{\langle q_0, f' \rangle} A_2$ *such that* $\boldsymbol{d}'_0$ *and* $\boldsymbol{d}'_1$ *are sequential-independent.*
2. *If both rules are mono-typed,* a_0, c *and* a_2 *are mono, and also* $L_0 \sqcap R_1 \sqsubseteq D_0 \sqcap D_1$ *in* $\mathrm{Sub}(T)$*, then* $C' = L_0 \sqcup (D_0 \sqcap D_1) \sqcup R_1$.

Proof. For the the first part of the theorem see [12, 8, 13]. For the second half, let $w_0 \colon R_0 \to D_1$ and $u_0 \colon L_1 \to D_0$ be such that $h = \gamma_1 \circ w_0$ and $g = \delta_0 \circ u_0$.

We obtain the following four diagrams: square (1) by pullback, also yielding pullbacks (2) and (3). Squares (4) and (5) by pushout, also yielding pushouts (6) and (7). Finally, square (8) by pushout. Notice that all the morphisms in the diagrams are mono.

$$\begin{array}{ccccc} L_0 & \xleftarrow{\alpha_0} & K_0 & \xrightarrow{\beta_0} & R_0 \\ {\scriptstyle u_1}\downarrow & (4) & \downarrow & (2) & \downarrow{\scriptstyle w_0} \\ E_0 & \leftarrowtail & D_0 \sqcap D_1 & \rightarrowtail & D_1 \\ {\scriptstyle \gamma'_1}\downarrow & (6) & \downarrow & (1) & \downarrow{\scriptstyle \gamma_1} \\ A_0 & \xleftarrow[\gamma_0]{} & D_0 & \xrightarrow[\delta_0]{} & C \end{array} \qquad \begin{array}{ccccc} L_1 & \xleftarrow{\alpha_1} & K_1 & \xrightarrow{\beta_1} & R_1 \\ {\scriptstyle u_0}\downarrow & (3) & \downarrow & (5) & \downarrow{\scriptstyle w_1} \\ D_0 & \leftarrowtail & D_0 \sqcap D_1 & \rightarrowtail & E_1 \\ {\scriptstyle \delta_0}\downarrow & (1) & \downarrow & (7) & \downarrow{\scriptstyle \delta'_0} \\ C & \xleftarrow[\gamma_1]{} & D_1 & \xrightarrow[\delta_1]{} & A_2 \end{array}$$

$$\begin{array}{ccccc} L_1 & \xleftarrow{\alpha_1} & K_1 & \xrightarrow{\beta_1} & R_1 \\ {\scriptstyle u_0}\downarrow & (3) & \downarrow & (5) & \downarrow{\scriptstyle w_1} \\ D_0 & \leftarrowtail & D_0 \sqcap D_1 & \rightarrowtail & E_1 \\ {\scriptstyle \gamma_0}\downarrow & (6) & \downarrow & (8) & \downarrow{\scriptstyle \gamma'_0} \\ A_0 & \xleftarrow[\gamma'_1]{} & E_0 & \xrightarrow[\delta'_1]{} & C' \end{array} \qquad \begin{array}{ccccc} L_0 & \xleftarrow{\alpha_0} & K_0 & \xrightarrow{\beta_0} & R_0 \\ {\scriptstyle u_1}\downarrow & (4) & \downarrow & (2) & \downarrow{\scriptstyle w_0} \\ E_0 & \leftarrowtail & D_0 \sqcap D_1 & \rightarrowtail & D_1 \\ {\scriptstyle \delta'_1}\downarrow & (8) & \downarrow & (7) & \downarrow{\scriptstyle \delta_1} \\ C' & \xleftarrow[\gamma'_0]{} & E_1 & \xrightarrow[\delta'_0]{} & A_2 \end{array}$$

Notice that $E_0 = L_0 \sqcup (D_0 \sqcap D_1)$ (because a_0 is mono) and $E_1 = R_1 \sqcup (D_0 \sqcap D_1)$ (since a_2 is mono). It remains to show that $C' = E_0 \sqcup E_1$ for which it suffices to show that $E_0 \sqcap E_1 = D_0 \sqcap D_1$. But by assumption $E_0 \sqcap E_1 = (L_0 \sqcap R_1) \sqcup (D_0 \sqcap D_1) = D_0 \sqcap D_1$. □

From a true concurrency point of view, we do not want to distinguish among derivations which differ only in the order of sequential-independent direct derivations. This is formalized by the relation introduced next.

Definition 10 (Derivation switching). Let $\langle \tau, \boldsymbol{d}^\tau \rangle$ be a derivation and assume that the direct derivations $\boldsymbol{d}^\tau_{m-1}$ and $\boldsymbol{d}^\tau_m$ are sequential-independent. Let τ' be the path obtained from τ by switching these two direct derivations according to Theorem 9. Finally let $\boldsymbol{d}^{\tau'}$ be a diagram of τ'. Then we say that the two derivations are *switchings* of each other and write $\langle \tau, \boldsymbol{d}^\tau \rangle \overset{sw}{\sim} \langle \tau', \boldsymbol{d}^{\tau'} \rangle$.

4 Occurrence Grammars

In this section we shall introduce the central notion of *occurrence grammar* which will be used to describe the computation of a system modulo concurrency and on which the notion of process—introduced in Definition 20—relies.

We begin by defining the *asymmetric conflict* relation. It arises in any computational formalism where resources can be read without being consumed. The notion of asymmetric conflict has been previously defined and used for similar purposes in the concrete cases of Petri nets and graph transformation systems. Note that in this paper we deal only with deterministic occurrence grammars.

In the general setting of adhesive grammars, asymmetric conflict can be defined using the rule relations of Definition 7: rules p, q are in asymmetric conflict (written $p \nearrow q$) whenever either p is a (possibly indirect) cause of q or p is disabled by q. In an occurrence grammar every rule occurs exactly once; thus p *must* be executed before q.

Definition 11 (Asymmetric conflict, (co-)causes). Let $\mathcal{G} = \langle\langle S, P, \pi\rangle, T, t\rangle$ be a mono-typed grammar. Then $\nearrow = <^+ \cup (\ll \setminus \mathrm{id}_P)$, where id_P is the identity relation on P, is called *asymmetric conflict*. For a subobject $A \in \mathrm{Sub}(T)$ we define

$$\llcorner A \lrcorner = \{q \in P \mid R_q \sqcap A \not\sqsubseteq K_q\} \quad \text{and} \quad \ulcorner A \urcorner = \{q \in P \mid L_q \sqcap A \not\sqsubseteq K_q\}$$

as the sets of *(direct) causes* and *(direct) co-causes of* A respectively.

We are now ready to define the notion of occurrence grammars. Technically an occurrence grammar is a grammar with special properties which generalizes the notions of deterministic occurrence nets [11] and grammars [4] defined in the setting of Petri nets and graph grammars, respectively.

Definition 12 (Occurrence grammars). A grammar $\mathcal{O} = \langle\langle S, P, \pi\rangle, T, t\rangle$ is a *pre-occurrence grammar* if it is mono-typed,

1. P is finite and the relation $\nearrow$ is acyclic,
2. the start object S has no causes, i.e. $\llcorner S \lrcorner = \varnothing$,
3. there are neither forward nor backward conflicts, i.e., for all $q \neq q' \in P$

$$(L_{q'} \sqcap L_q) \sqsubseteq K_{q'} \sqcup K_q \quad \text{and} \quad (R_{q'} \sqcap R_q) \sqsubseteq K_{q'} \sqcup K_q.$$

A pre-occurrence grammar $\mathcal{O}$ is said to be an *occurrence grammar* if also:

4. there is an *end object* $F \in \mathrm{Sub}(T)$ such that $\ulcorner F \urcorner = \varnothing$;
5. for all subobjects $A \in \mathrm{Sub}(T)$

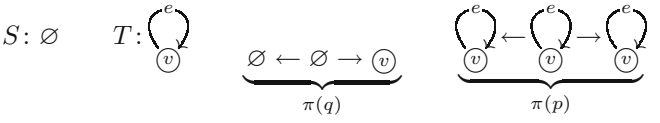
$$(a) \quad A \sqsubseteq \left(S \sqcup \bigsqcup_{q \in \llcorner A \lrcorner} R_q\right) \quad \text{and} \quad (b) \quad A \sqsubseteq \left(F \sqcup \bigsqcup_{q \in \ulcorner A \urcorner} L_q\right).$$

The requirements of Definition 12 above can be motivated as follows: First, $\nearrow$ must be acyclic, since there is otherwise no valid execution order for all rules of the occurrence grammar. Furthermore there are no forward conflicts, meaning that the occurrence grammar is deterministic, and no backward conflicts which roughly amounts to saying that "everything" is generated by at most one rule. It can be shown that S and F are uniquely determined by the axiom 5 of Definition 12.

Indeed, the axiom 5 is central for the following theory. It intuitively says that "everything" is either in the start object or generated at some point and that also the converse holds: "everything" is either in the end object or it is consumed at some time. The first part is needed to show that when we put the rules of an occurrence grammar into sequence according to asymmetric conflict and apply

an initial part of this sequence, we reach an object that contains the left-hand side of the next rule. Then the second part is needed to prove that also the pushout complement exists and thus the rule can actually be applied. (See also the proof of Theorem 19.) Its role is further explained by the example below.

Example 13 (Pre-occurrence grammar that is not an occurrence grammar). Consider the adhesive category of graphs and graph homomorphisms, **Graph**. Now take a grammar with the empty graph $\varnothing$ as start object S, and two rule names p, q with associated rules and type graph as shown below.



The typing is given by the obvious inclusions. This is clearly a pre-occurrence grammar, but not an occurrence grammar since axiom 5(a) of Definition 12 is violated. Indeed, $\lfloor T \rfloor = \{q\}$ and thus $T \not\sqsubseteq S \sqcup \bigsqcup_{q' \in \lfloor T \rfloor} R_{q'} = S \sqcup R_q = R_q$. Note that this corresponds to the fact that the graph obtained after applying q is *too small* to contain the left-hand side of p.

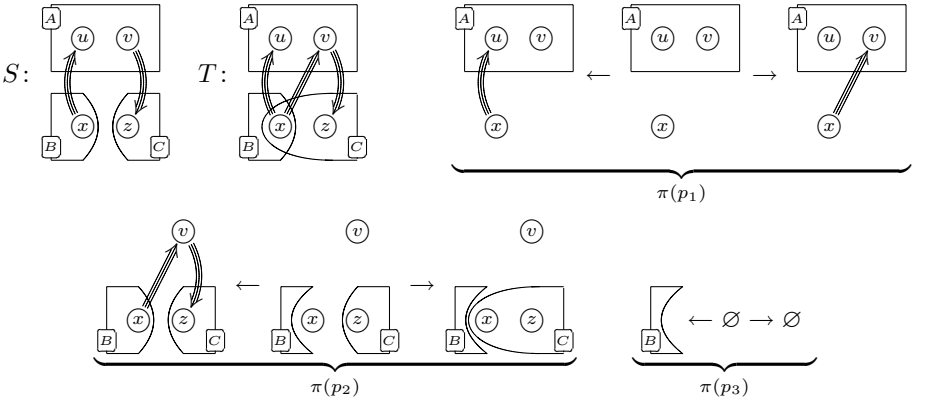
Similarly, when we consider the reversed pre-occurrence grammar (view rules from right to left) with T as the start object, axiom 5(b) of Definition 12 does not hold. In order to see this observe that now the end object is the empty graph and that only (the reversed) q is a co-cause for T, which leads to $T \not\sqsubseteq F \sqcup \bigsqcup_{q' \in \ulcorner A \urcorner} L_{q'}$. This is related to the fact that—after applying rule p (reversely) to T—q cannot be applied since the pushout complement for $\varnothing \rightarrowtail \circ \rightarrowtail \overset{\circ}{\bigcirc}$ does not exist, due to the presence of the edge.

In previous approaches, the subobject inclusions followed indirectly from axioms about individual items. For instance [5] defines a deterministic occurrence grammar $\mathcal{O}$ requiring that whenever a node v is deleted by a rule of $\mathcal{O}$ and an edge e attached to v is created by $\mathcal{O}$, then $\mathcal{O}$ must also delete e.

Example 14 (Graphs with scopes). In order to show that our theory applies to a setting wider than standard graph rewriting, we consider graphs with scopes where each node is contained in a set of scopes. These graphs can be viewed as objects of the functor category $\mathbf{Set}_{\mathbf{fin}}^{\bullet \leftarrow \bullet \rightarrow \bullet \leftarrow \bullet}$, which is adhesive. Concretely, every object consists of a set of nodes V, a set of edges E, a set of scopes S and an auxiliary set X, used to relate nodes and scopes. We have functions $src, tgt \colon E \to V$, $sc_S \colon X \to S$, $sc_V \colon X \to V$. If there is an element $x \in X$ with $sc_S(x) = s \in S$ and $sc_V(x) = v \in V$ we say that v is contained in or within scope s. A node may belong to several scopes and a scope may contain several nodes. We draw the graph part of the objects in the usual way. Scopes are depicted by labelled boxes around the nodes they contain (see below).

The following example grammar is inspired by scope extrusion in process calculi. We want to model that a node is moved from one scope into another by a reaction rule. The first rule (p_1) can move the target of an edge within the same scope, the second (p_2) is a reaction where a node v is transferred from one

scope to another whenever there is a two-edge path from it to a node w within the second scope, and the third (p_3) models garbage collection of empty scopes. Note that rule (p_3) cannot be applied to non-empty scopes, since the pushout complement of diagram (1) in Definition 4 would not exist, intuitively because the removal of the scope would leave some dangling links.



By taking S and T above as the start and type graph respectively, and the obvious inclusions as rule typings we obtain an occurrence grammar where p_1 is a cause for p_2 ($p_1 < p_2$) and p_2 is in asymmetric conflict with p_3 ($p_2 \nearrow p_3$).

After these motivating examples, we will continue to develop the theory. First we show that if every rule is applied at most once then the reached object is mono-typed. A consequence of this is that any object reachable in a consuming pre-occurrence grammar is mono-typed.

Proposition 15 (Quasi-safety and safety of consuming grammars). *Let $\mathcal{O} = \langle\langle S, P, \pi\rangle, T, t\rangle$ be a pre-occurrence grammar. Then for each path $\tau = \langle q_i, f_i\rangle_{i\in[n]}$ and typed derivation $\rho = \langle \tau, \boldsymbol{d}^\tau, c\rangle$, with $\boldsymbol{d}^\tau$ witnessing $S \overset{\tau}{\Rightarrow} A_n$, if no rule occurs twice in τ then*

1. *A_n is mono-typed, i.e., c_{A_n} is a mono,*
2. *asymmetric conflict is respected, i.e. $\forall i, j \in [n] \,.\, q_i \nearrow q_j \Rightarrow i < j$,*
3. *the inclusion cocone to $S \sqcup \bigsqcup_{i\in[m]} R_i$ for $m \leq n$ is a colimit of $\boldsymbol{d}^\tau_{[m]}$.*

In particular, if $\mathcal{O}$ is consuming then any rule can be applied at most once in each typed $\mathcal{O}$-derivation and thus 1–3 above hold for any typed derivation.

Another fact that holds in the setting of pre-occurrence grammars is that all typed derivations which apply the same rules, possibly in different order, are equivalent when seen as truly concurrent computations. Formally, this involves the notion of switch-equivalence for typed derivations.

Definition 16 (Switch equivalence). Let $\rho = \langle \tau, \boldsymbol{d}^\tau, c\rangle$ and $\rho' = \langle \tau', \boldsymbol{d}^{\tau'}, c'\rangle$ be two typed $\mathcal{G}$-derivations, with $\tau = \langle q_i, f_i\rangle_{i\in[n]}$ and $\tau' = \langle q'_i, f'_i\rangle_{i\in[n]}$. Then ρ and ρ' are *isomorphic*, written $\rho \cong \rho'$, if $q_i = q'_i$ for each $i \in [n]$ and there

is a diagram isomorphism $\iota\colon \langle \boldsymbol{d}^{\tau}, c\rangle \cong \langle \boldsymbol{d}^{\tau'}, c'\rangle$ that relates the start object, rule-occurrences and the type objects by identities.

Moreover $\rho \overset{sw}{\sim} \rho'$ if $\langle \tau, \boldsymbol{d}^{\tau}\rangle \overset{sw}{\sim} \langle \tau', \boldsymbol{d}^{\tau'}\rangle$ and finally *switch-equivalence* $\overset{sw}{\approx}$ is the union of the transitive closure of $\overset{sw}{\sim}$ and $\cong$, in signs $\overset{sw}{\approx} = (\overset{sw}{\sim})^* \cup \cong$.

Lemma 17 (Switch equivalence in pre-occurrence grammars). *Let* $\mathcal{O} = \big\langle \langle S, P, \pi\rangle, T, t\big\rangle$ *be a pre-occurrence grammar, and let* $\rho = \langle \tau, \boldsymbol{d}^{\tau}, c\rangle$ *and* $\rho' = \langle \tau', \boldsymbol{d}^{\tau'}, c'\rangle$ *be typed* $\mathcal{O}$*-derivations where* $\tau = \langle q_i, f_i\rangle_{i\in[n]}$ *and* $\tau' = \langle q'_i, f'_i\rangle_{i\in[n]}$ *are paths in which no rule occurs twice and* $\big\langle q_i\big\rangle_{i\in[n]}$ *is a permutation of* $\big\langle q'_i\big\rangle_{i\in[n]}$. *Then the two typed derivations are switch-equivalent, i.e.,* $\rho \overset{sw}{\approx} \rho'$.

The above facts about pre-occurrence grammars have a premise about the existence of some derivation. In the context of proper occurrence grammars we can single out sufficient conditions for the existence of derivations, which can be described in terms of asymmetric conflict $\nearrow$.

Definition 18 (Rule linearizations). Let $\mathcal{O} = \big\langle \langle S, P, \pi\rangle, T, t\big\rangle$ be a pre-process and let $P' \subseteq P$ and $n = |P'|$. Then a sequence $\boldsymbol{q} = \big\langle q_i\big\rangle_{i\in[n]} \in (P')^*$ is a *(rule) linearization of* P' if $P' = \{q_i \mid i \in [n]\}$ and $\forall i, j \in [n]\,.\, q_i \nearrow q_j \Rightarrow i < j$. The set of all linearizations of P' is denoted by $\mathrm{lin}(P')$ and $\boldsymbol{q}_i = q_i$ by convention.

We write $S \overset{\boldsymbol{q}}{\Rightarrow} A$ if $S \overset{\tau}{\Rightarrow} A$ and $\tau = \langle q_i, f_i\rangle_{i\in[n]}$ is a path for some sequence of matches $\big\langle f_i\big\rangle_{i\in[n]}$.

The next theorem gives two central results. Firstly, if $\mathcal{O}$ is an occurrence grammar, then there exists a typed derivation which rewrites the start object into the end object, applying all the rules in any order that respects asymmetric conflict. Secondly, if the type object is not too large and there exists a linearization of all rules that leads to a typed derivation then a pre-occurrence grammar is an occurrence grammar.

Theorem 19. *Let* $\mathcal{O}$ *be a pre-occurrence grammar.*

1. *If* $\mathcal{O}$ *is an occurrence grammar, then* $\forall \boldsymbol{q} \in \mathrm{lin}(P).\, S \overset{\boldsymbol{q}}{\Rightarrow} F$, *where* F *is the end object of* $\mathcal{O}$.
2. *If* $\exists \boldsymbol{q} \in \mathrm{lin}(P).\, \exists F \in \mathrm{Sub}(T).\, S \overset{\boldsymbol{q}}{\Rightarrow} F$ *and* $T = S \sqcup \bigsqcup_{q\in P} R_q$, *then* $\mathcal{O}$ *is an occurrence grammar.*

Proof (idea). The crucial point is the proof of the first part, i.e., of the fact that any linearization of P gives rise to a typed derivation. Let $\boldsymbol{q} = \boldsymbol{p}q\boldsymbol{p}' \in \mathrm{lin}(P)$ and assume that $S \overset{\boldsymbol{p}}{\Rightarrow} A$. Then we have to show that $L_q \sqsubseteq A$ and that the pushout complement for $A \leftarrowtail L_q \overset{\alpha}{\leftarrowtail} K_q$ exists.

By using axiom 5(a) of Definition 12 we can prove that A is the greatest object with causes in $\boldsymbol{p}$ and co-causes in $\boldsymbol{p}'$. Then $L_q \sqsubseteq A$ follows immediately. It remains to show that the pushout complement exists: the candidate is $\tilde{D} = (S \sqcup \bigsqcup_{q\in\boldsymbol{p}} R_q) \sqcap (F \sqcup \bigsqcup_{q\in\boldsymbol{p}'} L_q)$.

By using only facts about pre-occurrence grammars one can show that $\tilde{D}$ is the greatest subobject of A which forms a pullback together with the arrows $A \leftarrowtail L_q \overset{\alpha}{\leftarrowtail} K_q$. Finally, using axiom 5(b) of Definition 12 and some elementary category theory we can show that $\tilde{D}$ is actually a pushout complement. □

An interesting point of the proof is that the question about the existence of pushout complements can be answered in lattice-theoretic terms only.

5 From Derivations to Processes and Back

We now come to the one-to-one correspondence between switch-equivalence classes of derivations and processes. After introducing the notion of process (for a given grammar), we show that such a process can be seen as a representative of a full class of switch-equivalent typed derivations, all of which are linearizations of the process. Vice versa, given a derivation, a colimit-based construction allows to derive a corresponding process. The result states that these two constructions are (essentially) inverse to each other.

We shall now define the notion of process, i.e., a truly concurrent computation of a specific grammar $\mathcal{G}$ represented by an occurrence grammar.

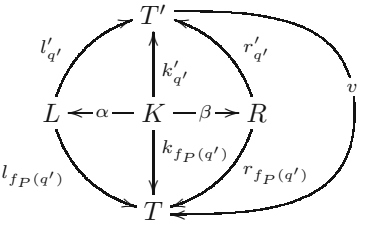
Definition 20 (Processes). Let $\mathcal{G} = \langle\langle S, P, \pi\rangle, T, t\rangle$ be a grammar. Then a *$\mathcal{G}$-process* is a triple $\mathcal{P} = \langle \mathcal{O}, v, f_P\rangle$ where $\mathcal{O} = \langle\langle S', P', \pi'\rangle, T', t'\rangle$ is an occurrence grammar and

- $v\colon T' \to T$ is a morphism between the type objects, and
- $f_P\colon P' \cup \{\underline{S'}\} \to P \cup \{\underline{S}\}$ is a function between rule names with $f_P(\underline{S'}) = \underline{S}$

such that for all $q' \in P' \cup \{\underline{S'}\}$

1. $\pi'(q') = \pi(f_P(q'))$
2. and[2] $v \circledcirc t'(q') = t(f_P(q'))$

i.e. the diagram on the right commutes, where $\pi'(q') = L \overset{\alpha}{\leftarrowtail} K \overset{\beta}{\rightarrowtail} R = \pi(f_P(q'))$.



Let $\mathcal{P}_1$ and $\mathcal{P}_2$ be two $\mathcal{G}$-processes. An *isomorphism* $\langle i, j\rangle\colon \mathcal{P}_1 \cong \mathcal{P}_2$ from $\mathcal{P}_1$ to $\mathcal{P}_2$ is a pair $\langle i, j\rangle$ such that (i) $\langle \mathcal{O}_1, i, j\rangle$ is an $\mathcal{O}_2$-process, (ii) $i\colon T_1 \to T_2$ is an isomorphism satisfying $v_2 \circ i = v_1$, and (iii) $j\colon P_1 \cup \{\underline{S_1}\} \to P_2 \cup \{\underline{S_2}\}$ is a bijection satisfying $f_{P1} = f_{P2} \circ j$.

Intuitively, an occurrence grammar $\mathcal{O}$ only represents an "autonomous" concurrent computation, whereas the pair $\langle v, f_P\rangle$ provides a link back to a grammar. The morphism v specifies how such a computation can be "typed" over the type object of $\mathcal{G}$, and f_P specifies how the rule occurrences of $\mathcal{O}$ can be seen as instances of rules in $\mathcal{G}$.

[2] For a cocone c to an object A and a morphism $v\colon A \to B$ we denote by $v \circledcirc c$ the cocone to B obtained by composing every morphism in c with v.

Given a process $\mathcal{P}$ of a grammar $\mathcal{G}$, we can obtain a corresponding typed derivation in $\mathcal{G}$ by taking any linearization of the rules in $\mathcal{O}$, applying each such rule in the specified order (possible by Theorem 19) and retyping the generated derivation over the type object of $\mathcal{G}$.

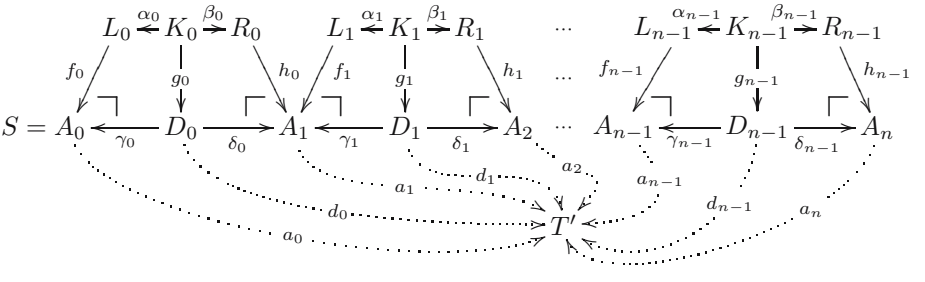
Definition 21 (Drv—**derivations of a process).** Let $\mathcal{P} = \langle \mathcal{O}, v, f_P \rangle$ be a $\mathcal{G}$-process, where $\mathcal{O} = \langle\langle S, P, \pi\rangle, T', t'\rangle$. Let $\boldsymbol{q} \in \mathrm{lin}(P)$ be a linearization of P and let $\rho = \langle \tau, \boldsymbol{d}^\tau, c\rangle$ be a typed derivation witnessing $S \overset{\boldsymbol{q}}{\Rightarrow}_{\mathcal{O}} F$. Then $\langle \tau, \boldsymbol{d}^\tau, v \circledcirc c\rangle$ is called a typed $\mathcal{P}$*-derivation*. The set of all such derivations is denoted by $\mathrm{Drv}(\mathcal{P})$.

The next proposition shows that all derivations of a given process are "equivalent" from a true concurrency point of view. Hence Drv induces a mapping from (isomorphism classes of) processes to switch-equivalence classes of derivations.

Proposition 22. *Let $\mathcal{P}$ and $\mathcal{P}'$ be processes such that $\mathcal{P} \cong \mathcal{P}'$. Then for all $\rho \in \mathrm{Drv}(\mathcal{P})$ and $\rho' \in \mathrm{Drv}(\mathcal{P}')$ it holds $\rho \overset{sw}{\approx} \rho'$.*

Vice versa, given any derivation in a grammar $\mathcal{G}$, we can generate a corresponding process as follows. The colimit of the (untyped part) of the derivation diagram is the type object of the process, while the rule instances of the derivation become the rules of the process. The morphism back to the type object of $\mathcal{G}$ is given by the mediating morphism to the typed derivation cocone. The next definition describes this procedure formally.

Definition 23 (Prc—**processes of a derivation).** Let $\tau = \langle q_i, f_i\rangle_{i\in[n]}$ be a path and $\rho = \langle \tau, \boldsymbol{d}^\tau, c\rangle$ be a typed derivation for a grammar $\mathcal{G} = \langle\langle S, P, \pi\rangle, T, t\rangle$. Let $\bar{c}$ be a colimit cocone for $\boldsymbol{d}^\tau$ to T', whose components are the dotted arrows below.



Define $\mathcal{O} = \langle\langle S', P', \pi'\rangle, T', t'\rangle$ to be a grammar where:

- $S' = S$;
- $P' = \{\langle q_i, i\rangle \mid i \in [n] \wedge \tau_i = \langle q_i, f_i\rangle\}$ is a set that contains a *rule occurrence name* for each rule occurrence of $\boldsymbol{d}^\tau$, and
- π' with $\pi'(\langle q_i, i\rangle) = \pi(q_i)$ assigns each rule occurrence name the rule of the grammar $\mathcal{G}$ it originates from; and
- $t'(\langle q_i, i\rangle)$ is a cocone for $\pi(q_i)$ to T', which gives the typing for each rule occurrence $\langle q_i, i\rangle \in P'$ as indicated below

$$\left.\begin{matrix}\pi'(\langle q_i, i\rangle)\\ t'(\langle q_i, i\rangle)\end{matrix}\right\} \quad \begin{array}{ccccc} L_i & \xleftarrow{\alpha_i} & K_i & \xrightarrow{\beta_i} & R_i \\ & {\scriptstyle a_i \circ f_i}\searrow & \downarrow{\scriptstyle d_i \circ g_i} & \swarrow{\scriptstyle a_{i+1} \circ h_i} & \\ & & T' & & \end{array}$$

and $t'(\underline{S'})$ is the cocone obtained by taking three times morphism a_0.

Finally let $v\colon T' \to T$ be the mediating morphism from the colimit $\bar{c}$ to the cocone c. Then

$$\mathcal{P} = \langle \mathcal{O}, v, f_P\colon P' \uplus \{\underline{S'}\} \to P \uplus \{\underline{S}\}\rangle$$

with $f_P(\langle q_i, i\rangle) = q_i$ and $f_P(\underline{S'}) = \underline{S}$ is a *ρ-process*. The set of all ρ-processes—all of them being isomorphic to each other—is denoted by $\mathrm{Prc}(\rho)$.

The next proposition shows that starting from switch-equivalent derivations, the construction described in Definition 23 produces isomorphic processes. Hence Prc can be seen as a function from switch-equivalence classes of derivations to isomorphism classes of processes.

Proposition 24. *Let ρ and ρ' be typed $\mathcal{G}$-derivations such that $\rho \overset{sw}{\approx} \rho'$. Then $\mathcal{P} \cong \mathcal{P}'$ holds for all $\mathcal{P} \in \mathrm{Prc}(\rho)$ and $\mathcal{P}' \in \mathrm{Prc}(\rho')$.*

We conclude with the main result of this section, stating that Prc and Drv can be seen as functions between switch-equivalence classes of derivations and isomorphism classes of processes, and that they are inverse to each other.

Theorem 25. *Let ρ be a typed $\mathcal{G}$-derivation and $\mathcal{P}$ be a $\mathcal{G}$-process. Then:*

1. $\rho' \in \mathrm{Drv}\big(\mathrm{Prc}(\rho)\big)$ *implies* $\rho' \overset{sw}{\approx} \rho$
2. $\mathcal{P}' \in \mathrm{Prc}\big(\mathrm{Drv}(\mathcal{P})\big)$ *implies* $\mathcal{P}' \cong \mathcal{P}$

6 Conclusion

We have shown that the notion of process, originally introduced for Petri nets, can be studied in the general setting of DPO rewriting systems over adhesive categories. This is theoretically pleasing, since it allows one to study this fundamental concept at the same abstract level as, for instance, the notion of sequential-independence.

While the fact that processes can be studied in an abstract framework may not seem surprising, the generalization is non-trivial to obtain. The reason is that the previous definitions of occurrence grammars and processes, e.g. of Petri nets and graph grammars, used the inherently set-theoretical concept of items: atomic units that are consumed and produced. The absence of an analogous concept for adhesive categories has required the development of original techniques, mainly relying on the algebra of the subobject lattice of the type object.

As a consequence of its generality, the theory developed in this paper is applicable to a wide range of rewriting systems. It enables us to handle various graph-like structures which appear in the literature and are used in tools.

While starting the development of an encompassing theory of true concurrency, we have also laid the foundations for the use of partial order verification techniques. Specifically, the generalization of methods developed for Petri nets and graph transformation systems (see, e.g., [14, 10, 2, 3]) appears as a stimulating direction of research. In order to achieve this goal, future work will concern *unfoldings*: non-deterministic (infinite) processes which fully describe the behavior of a system.

References

1. P. Baldan. *Modelling Concurrent Computations: from Contextual Petri Nets to Graph Grammars*. PhD thesis, Dipartimento di Informatica, Università di Pisa, 2000.
2. P. Baldan, A. Corradini, and B. König. A static analysis technique for graph transformation systems. In *Proc. of CONCUR '01*, volume 2154 of *LNCS*, pages 381–395. Springer Verlag, 2001.
3. P. Baldan, A. Corradini, and B. König. Verifying finite-state graph grammars: an unfolding-based approach. In *Proc. of CONCUR 2004*, volume 3170 of *LNCS*, pages 83–98. Springer Verlag, 2004.
4. P. Baldan, A. Corradini, and U. Montanari. Concatenable graph processes: relating processes and derivation traces. In *Proc. ICALP'98*, volume 1443 of *LNCS*. Springer Verlag, 1998.
5. P. Baldan, A. Corradini, and U. Montanari. Unfolding and Event Structure Semantics for Graph Grammars. In *Proc. of FoSSaCS '99*, volume 1578 of *LNCS*, pages 73–89. Springer Verlag, 1999.
6. A. Corradini, U. Montanari, and F. Rossi. Graph processes. *Fundamenta Informaticae*, 26:241–265, 1996.
7. H. Ehrig. Introduction to the Algebraic Theory of Graph Grammars. In *Proceedings of the 1st International Workshop on Graph-Grammars and Their Application to Computer Science and Biology*, volume 73 of *LNCS*, pages 1–69. Springer Verlag, 1979.
8. H. Ehrig, A. Habel, H.-J. Kreowski, and F. Parisi-Presicce. Parallelism and concurrency in high-level replacement systems. *Mathematical Structures in Computer Science*, 1:361–404, 1991.
9. H. Ehrig, A. Habel, J. Padberg, and U. Prange. Adhesive high-level replacement categories and systems. In *Proc. of ICGT'04*, volume 3256 of *LNCS*, pages 144–160. Springer Verlag, 2004.
10. J. Esparza, S. Römer, and W. Vogler. An improvement of McMillan's unfolding algorithm. *Formal Methods in System Design*, 20(20):285–310, 2002.
11. U. Goltz and W. Reisig. The non-sequential behaviour of Petri nets. *Information and Control*, 57:125–147, 1983.
12. A. Habel, J. Müller, and D. Plump. Double-pushout graph transformation revisited. *Mathematical Structures in Computer Science*, 11(5):637–688, 2001.
13. S. Lack and P. Sobociński. Adhesive and quasiadhesive categories. *Theoretical Informatics and Applications*, 39(2):511–546, 2005.
14. K.L. McMillan. *Symbolic Model Checking*. Kluwer, 1993.