Dynamic Graph Transformation Systems*

Roberto Bruni[1] and Hernán Melgratti[2]

[1] Computer Science Department, University of Pisa, Italy.
[2] IMT Lucca Institute for Advanced Studies, Italia.
bruni@di.unipi.it, hernan.melgratti@imtlucca.it

Abstract. We introduce an extension of Graph Grammars (GGs), called Dynamic Graph Grammars (DynGGs), where the right-hand side of a production can spawn fresh parts of the type graph and fresh productions operating on it. The features of DynGGs make them suitable for the straightforward modeling of reflexive mobile systems like dynamic nets and the Join calculus. Our main result shows that each DynGG can be modeled as a (finite) GG, so that the dynamically generated structure can be typed statically, still preserving exactly all derivations.

1 Introduction

Graphs can model complex systems at a level of abstraction that is both intuitive and formal. Graph Grammars (GGs) originated in the late 60's as a suitable extension of string grammars: string concatenation is replaced by graph gluing and string rewriting by subgraph replacing. As a model of concurrency, there is also a close analogy between GGs and Petri nets (PNs), as a Petri net can be straightforwardly modeled as a particular GG over discrete graphs. Solid theoretical basis are now available for many different kinds of graph transformation, ranging from the essential node replacement systems [8] and edge replacement systems [6] to the more sophisticated synchronized hyperedge replacement systems [3,11,10] and algebraic approaches to graph rewriting [4,7].

To the best of our knowledge, one extension that has not been deeply investigated in the literature is the use of reflexive productions that can release new rewrite rules. Reflexive systems arise naturally in many areas where graph transformation techniques have been applied with success, like biological and chemical systems and distributed and mobile computing. Moreover, the reflexive extension of many different kinds of rewrite systems have been studied in the literature. In particular, dynamic nets are a mobile extension of PNs, expressive enough to model mobile calculi like π-calculus and Join calculus [1,2]. Dynamic nets are indeed strictly more expressive than PNs.

Exploiting the analogy between PNs and GGs, we propose a reflexive extension of GGs, called *Dynamic Graph Grammars* (DynGGs), whose generality is witnessed by encodings of dynamic nets and Join calculus. However, when posing the question:

"Are Dynamic Graph Grammars more expressive than ordinary ones?"

our main result provides a negative answer: though DynGGs can offer a more convenient abstraction, computationally speaking they are not more expressive than GGs.

* Research supported by the EU FET-GC2 IST-2004-16004 Integrated Project SENSORIA

From GGs to DynGGs. To complete this informal introduction, we sketch the design choices of DynGGs, introduce some terminology and give a minimal example.

Let T denote a type graph and G_T a graph typed over T. Ordinary T-typed DPO productions are spans of the form $p : (L_T \stackrel{l}{\leftarrow} K_T \stackrel{r}{\rightarrow} R_T)$ such that l and r preserve the typing of items in K_T. (We assume that the reader has some familiarity with GGs, and hence postpone exact formalization to later sections.)

A first simple extension is to consider productions like $p : (L_T \stackrel{l}{\leftarrow} K_T \stackrel{r}{\rightarrow} R_{T'})$ where $T \subseteq T'$, so that fresh types can be generated and used for typing $R_{T'}$. This way the fresh types introduced by p cannot be exploited in the (left-hand side of) productions. The idea is then to spawn also new productions able to operate on items typed in $T' \setminus T$.

A DynGG is thus a triple (T, G_T, P) where T is a type graph, G_T a graph typed over T and P a set of T-typed dynamic productions that take the form $p : (L_T \stackrel{l}{\leftarrow} K_T \stackrel{r}{\rightarrow} \mathcal{G}_{T'})$ where $\mathcal{G}_{T'}$ is again a (T'-typed) DynGG and r relates K_T to the initial graph of $\mathcal{G}_{T'}$. For example, when $P = \emptyset$, then the grammar is roughly a T-typed graph G_T, which is statically fixed and cannot change. Such grammars are called *static*. A production p is *static* if its right-hand side $\mathcal{G}_{T'}$ is a static grammar and $T' = T$. If all productions are static, then the grammar is called *shallow* and is essentially an ordinary GGs: the application of any production can neither change the type graph nor spawn new rules.

Figures 1 and 2 introduce a small ad hoc example whose purpose is to expose a peculiarity of dynamic rewrites. Let T_a be the singleton type graph with just one node a. Let $T_g \supset T_a$ consisting of nodes a and b and two edges $f : a \to b$, and $g : b \to b$.

Take the T_a-typed dynamic grammar $\mathcal{G}_a$ with the dynamic production p in Figure 1. For simplicity, we take the inclusions as legs of the span and draw the typing (dotted lines) only once for each item. The left-hand side (i) of p consists of a T_a-typed graph with just one node n_1, which is preserved by the context (ii), and the right-hand side (iii) spawns a shallow T_g-typed grammar $\mathcal{G}_p$ whose initial graph has, beside n_1, one additional node n_2 and one arc h. The grammar $\mathcal{G}_p$ itself has just one static production $q \in P_p$, illustrated in Figure 2: the left-hand side (i) is a graph with three nodes m_0 and m_1, m_2, which are all preserved (ii), and two arcs h_1, h_2, which are deleted, and the right-hand side (iii) spawns the static T_g-typed grammar $\mathcal{G}_q$ (i.e., a graph) with one additional arc l from m_1 to m_2 and type g.

Assume the initial graph of $\mathcal{G}_a$ is a discrete graph G_0 with one node k typed over a. The application of the production p (with the obvious matching from n_1 to k) spawns a fresh instance $\mathcal{G}_p^1$ of $\mathcal{G}_p$: the type graph becomes T_{g_1} and the underlying graph becomes $G_1 \supset G_0$ with nodes k (typed a) and k_1 (typed b_1) connected by an arc h_1 (typed f_1). Moreover, a production q_1 is now available beside p. A second application of p spawns another fresh instance $\mathcal{G}_p^2$ of $\mathcal{G}_p$: the type graph becomes $T_{g_1} \cup T_{g_2}$ and the underlying graph becomes $G_2 \supset G_1$ with a new node k_2 and an arc $h_2 : k \to k_2$. Again, a production q_2 is now available. Similarly, p can be applied again and again (see Figure 6). However, no suitable matching can ever be found for the application of q_1, q_2, etc. In fact, it is not possible to find two arcs with the same type, say f_i, and the identification condition prevents a non-injective matching of the two arcs in the left-hand side of q_i.

Now compare $\mathcal{G}_a$ with its static, flattened T_g-typed version, where only two productions p and q are available at any time: after two derivation steps with p the underlying graph has two nodes typed over b and two arcs typed over f, and thus q can be applied!

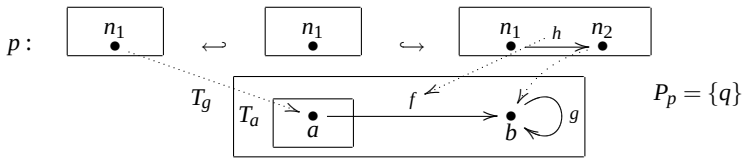


Fig. 1. A dynamic production p.

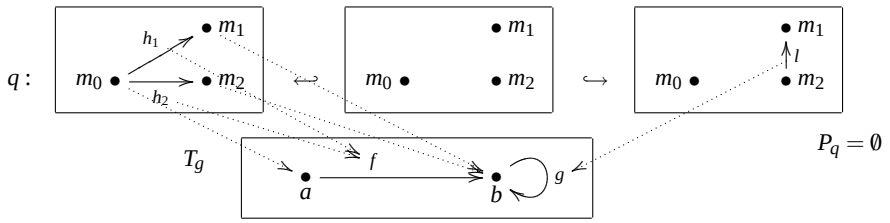


Fig. 2. A static production q.

The difference lies in the *separation principle* that DynGGs imposes on items produced as freshly-typed instances by different applications of the same production. We shall reprise this example to show that incautious encodings of DynGGs intro GGs would introduce unwanted derivations.

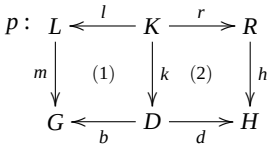
Synopsis. § 2 accounts for some basics of typed GGs [5]. The definition of DynGGs is original to this contribution and described in § 3, where DynGGs are shown to be a conservative extension of GGs. § 4 reports some sample encodings of other reflexive frameworks. The main result of the paper is in § 5, where it is shown that GGs have the same expressive power as DynGGs. Concluding remarks and future work are in § 6.

2 Typed Graph Grammars

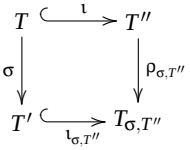
A *(directed) graph* is a tuple $G = \langle N_G, E_G, s_G, t_G \rangle$, where N_G is a set of *nodes*, E_G is a set of *edges* (or *arcs*), and $s_G, t_G : E_G \to N_G$ are the *source* and *target* functions. We shall omit subscripts when obvious from the context. A *graph morphism* $f : G \to G'$ is a couple $f = \langle f_N : N \to N', f_E : E \to E' \rangle$ such that: $s' \circ f_E = f_N \circ s$ and $t' \circ f_E = f_N \circ t$.

Definition 2.1 (Typed Graph). *Given a* graph of types T, *a* T-typed graph *is a pair* $\langle |G|, \tau_G \rangle$, *where* $|G|$ *is the* underlying *graph and* $\tau_G : |G| \to T$ *is a total morphism.*

In GGs the graph $|G|$ defines the configuration of the system and its items (nodes and edges) model resources, while τ_G defines the *typing* of the resources. Hence, the underlying graph $|G|$ evolves dynamically, while the type graph T is statically fixed and cannot change at run-time. For example, when encoding Petri nets in GGs the places form the discrete graph of types, while markings form the configurations of the system.



(a) A DPO direct derivation.



(b) Standard type graph construction.

Fig. 3. DPO and type graph construction.

A *morphism* between T-typed graphs $f : G_1 \to G_2$ is a graph morphisms $f : |G_1| \to |G_2|$ such that $\tau_{G_1} = \tau_{G_2} \circ f$. The category of T-typed graphs and their morphisms is denoted by T-**Graph**. Since we work only with typed notions, we will usually omit the qualification "typed", and we will not indicate explicitly the typing morphisms. The following notion of retyping will be used extensively in the context of DynGGs.

Definition 2.2 (Graph Retyping). *Given a T-typed graph $G = \langle |G|, \tau_G \rangle$ and a morphism $\sigma : T \to T'$ we denote by $\sigma \cdot G$ the T'-typed graph $\sigma \cdot G = \langle |G|, \sigma \circ \tau_G \rangle$.*

The key notion to *glue* graphs together is that of a categorical pushout. Roughly, a pushout pastes two graphs by injecting them in a larger graph that is (isomorphic to) their disjoint union modulo the collapsing of some common part. We recall that a *span* is a pair (b,c) of morphisms $b : A \to B$ and $c : A \to C$. A *pushout* of the span (b,c) is then an object D together with two (co-final) morphisms $f : B \to D$ and $g : C \to D$ such that: (i) $f \circ b = g \circ c$ and (ii) for any other choice of $f' : B \to D'$ and $g' : C \to D'$ s.t. $f' \circ b = g' \circ c$ there is a unique $d : D \to D'$ s.t. $f' = d \circ f$ and $g' = d \circ g$. If the pushout is defined, then c and g is called the *pushout complement* of $\langle b, f \rangle$.

A *(T-typed graph) DPO production* $p : (L \stackrel{l}{\leftarrow} K \stackrel{r}{\to} R)$ is a span of injective graph morphisms $l : K \to L$ and $r : K \to R$. The T-typed graphs L, K, and R are called the *left-hand side*, the *interface*, and the *right-hand side* of the production, respectively.

Definition 2.3 (DPO graph grammar). *A (T-typed) DPO graph grammar $\mathcal{G}$ is a tuple $\langle T, G_{in}, P \rangle$, where G_{in} is the* initial (T-typed) graph, *P is a set of DPO* productions.

Given a graph G, a production $p : (L \stackrel{l}{\leftarrow} K \stackrel{r}{\to} R)$, and a *match* $m : L \to G$, a *direct derivation δ from G to H using p (based on m)* exists, written $\delta : G \Rightarrow_p H$, if and only if the diagram in Figure 3(a) can be constructed, where both squares are pushouts in T-**Graph**: (1) the rewriting step removes from the graph G the items $m(L - l(K))$, yielding the graph D (with k, b as a pushout complement of $\langle m, l \rangle$); (2) then, fresh copies of the items in $R - r(K)$ are added to D yielding H (as a pushout of (k,r)). The interface K specifies both what is preserved and how fresh items must be glued to the existing part. The existence of the pushout complement of $\langle m, l \rangle$ is subject to the satisfaction of the following *gluing conditions* [4]:

- *identification condition:* $\forall x, y \in L$ if $x \neq y$ and $m(x) = m(y)$ then $x, y \in l(K)$;
- *dangling condition:* no arc in $G \setminus m(L)$ should be incident to a node in $m(L \setminus l(K))$.

The identification condition is satisfied by *valid matches*: a match is not valid if it requires a single item to be consumed twice, or to be both consumed and preserved.

A *derivation* is a sequence $\gamma = \{\delta_i : G_{i-1} \Rightarrow_{p_{i-1}} G_i\}_{i\in\{1,\dots,n\}}$ of direct derivations.

3 Dynamic Graph Grammars

As aforementioned, a T-typed dynamic production takes the form: $p : (L_T \xleftarrow{l} K_T \xrightarrow{r} \mathcal{G}_{T'})$ where $\mathcal{G}_{T'}$ is a suitable (T'-typed) dynamic graph grammar. A dynamic graph grammar can contain any number of such productions. Formally:

Definition 3.1 (Dynamic Graph Grammars). *The domain of dynamic graph grammars can be expressed as the least set* DGG *satisfying the equation:*

$$\begin{aligned}\mathcal{D} = \{(T, G_{in}, P) \mid\ & G_{in} \in \mathbf{Graph}_T \wedge \\ & \forall p : (L_T \xleftarrow{l} K_T \xrightarrow{r} \mathcal{G}_{T_p}) \in P.(\, L_T, K_T \in \mathbf{Graph}_T \wedge T \subset T_p \wedge \\ & \qquad \mathcal{G}_{T_p} = (T_p, G_{T_p}, P_p) \in \mathcal{D}\,) \quad \}\end{aligned}$$

where $\mathbf{Graph}_T$ *is the set of T-typed graphs and $r : \iota \cdot K_T \to G_{T_p}$ is a morphism between T_p-typed graphs, where $\iota : T \hookrightarrow T_p$ denotes the obvious sub-graph injection.*

Any element $\mathcal{G} = (T, G_{in}, P) \in$ DGG *is called a* Dynamic Graph Grammar. *It is* static *if $P = \emptyset$. It is* shallow *if $T = T_p$ and $\mathcal{G}_{T_p}$ is static for all $p : (L_T \xleftarrow{l} K_T \xrightarrow{r} \mathcal{G}_{T_p}) \in P$.*

Note that all Dynamic Graph Grammars are well-founded: since we take DGG as the least set satisfying the recursive domain equation above, the type graphs syntactically appearing in $\mathcal{G} = (T, G_{in}, P) \in$ DGG form a finite tree $\mathbb{T}(\mathcal{G})$ rooted in T, with parent relation given by immediate subsetting (i.e., T_i is parent of T_j iff $T_i \subset T_j$ and no T_k appears in $\mathcal{G}$ such that $T_i \subset T_k \subset T_j$) and where leaves are associated with static grammars. We remark that each type graph $T_p \supset T$ extends T with local declarations $T_p \setminus T$, whose scope is bounded by the specific production p. For simplicity, but without loss of generality, we assume that all additional items introduced by different type graphs inside $\mathcal{G}$ are named differently (i.e., each additional item occurs only in one type graph). We let $\mathbf{T}(\mathcal{G}) = \bigcup_{T_i \in \mathbb{T}(\mathcal{G})} T_i$ denote the *overall flat type graph* of $\mathcal{G}$, and let $\iota_{T_i} : T_i \hookrightarrow \mathbf{T}(\mathcal{G})$ denote the obvious sub-graph inclusion. Note that, by the structuring of $\mathbb{T}(\mathcal{G})$, the type graph $\mathbf{T}(\mathcal{G})$ is just the union of all the leaves of $\mathbb{T}(\mathcal{G})$.

Similarly, all nested productions in $\mathcal{G}$ form the tree $\mathbb{P}(\mathcal{G})$ rooted in P with parent relation given by immediate inclusion (i.e., the set of productions P_i is the parent of P_p iff $p : (L_T \xleftarrow{l} K_T \xrightarrow{r} (T_p, G_{T_p}, P_p)) \in P_i$). Given a T-typed dynamic production $p : (L_T \xleftarrow{l} K_T \xrightarrow{r} \mathcal{G}_{T_p})$ with $\mathcal{G}_{T_p} = (T_p, G_{T_p}, P_p)$ we say that the ordinary $\mathbf{T}(\mathcal{G})$-typed production $\mathsf{flat}(p) : (\iota_T \cdot L_T \xleftarrow{l} \iota_T \cdot K_T \xrightarrow{r} \iota_{T_p} \cdot G_{T_p})$ is the *flattening of p*. We let $\mathbf{P}(\mathcal{G}) = \bigcup_{P_i \in \mathbb{P}(\mathcal{G})} \{\mathsf{flat}(p) \mid p \in P_i\}$ denote the *overall set of flat productions* of $\mathcal{G}$. The $\mathbf{T}(\mathcal{G})$-typed shallow grammar $\mathbf{F}(\mathcal{G}) = (\mathbf{T}(\mathcal{G}), \iota_T \cdot G_{in}, \mathbf{P}(\mathcal{G}))$ is called the *flattening of* $\mathcal{G}$.

To define the dynamics of DynGGs we need a more advanced notion of retyping, which can be used to generate fresh items in the type graph. In the following, when considering type graph constructions, we assume that a standard choice of pushout objects

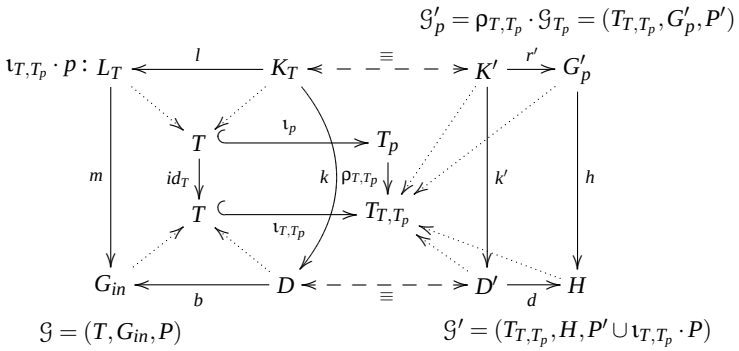


Fig. 4. A direct dynamic derivation.

satisfying the following requirements is available: Let $T \subset T''$ and $\sigma : T \to T'$ injective, then we denote by $T_{\sigma,T''}$ the pushout object of the inclusion $\iota : T \hookrightarrow T''$ and σ such that T' embeds in $T_{\sigma,T''}$ via set-theoretical inclusion $\iota_{\sigma,T''}$, while T'' embeds via an injection $\rho_{\sigma,T''}$ (see Figure 3(b)) that renames items in $T'' \setminus T$ with fresh names. When $T \subseteq T'$ and σ is the inclusion we replace the subscripting $(-)_{\sigma,T''}$ with $(-)_{T',T''}$.

Definition 3.2 (Fresh Graph Retyping). *Let $T \subset T''$. Given a T''-typed graph $G = \langle |G|, \tau_G \rangle$ and an injection $\sigma : T \to T'$ we let $\sigma \cdot G = \langle |G|, \rho_{\sigma,T''} \circ \tau_G \rangle$.*

Definition 3.3 (Dynamic Retyping). *Given a T-typed Dynamic Graph Grammar $\mathcal{G} = (T, G_{in}, P)$ and an injective morphism $\sigma : T \to T'$ we denote by $\sigma \cdot \mathcal{G}$ the T'-typed grammar defined recursively by letting $\sigma \cdot \mathcal{G} = (T', \sigma \cdot G_{in}, \sigma \cdot P)$, with $\sigma \cdot P = \{\sigma \cdot p \mid p \in P\}$ where $\sigma \cdot p : (\sigma \cdot L_T \stackrel{l}{\leftarrow} \sigma \cdot K_T \stackrel{r}{\rightarrow} \rho_{\sigma,T_p} \cdot \mathcal{G}_{T_p})$ for any $p : (L_T \stackrel{l}{\leftarrow} K_T \stackrel{r}{\rightarrow} \mathcal{G}_{T_p})$.*

Note that $\rho_{\sigma,T_p} \cdot \mathcal{G}_{T_p}$ is a T_{σ,T_p}-typed grammar and $\sigma \cdot p$ is a T'-typed production.

To define the behaviour of DynGGs, note that the type graph and the available productions can change over time: as the computation progresses new items and productions can be spawn. Hence, as it is typical of reflexive systems, the actual configuration must comprise data (i.e., the underlying graph), their typing and the control (i.e., available productions). This means that configurations are themselves DynGGs.

In DynGGs, productions are nested inside (the right-hand sides of) other productions, but only top-level productions can be applied, by finding a matching of their left-hand sides into the initial graph. When such a production p is applied, then fresh instances of the productions P_p, nested one level below p, become available at the top-level, and can be unwound themselves in successive steps. Given a DynGG $\mathcal{G} = (T, G_{in}, P)$, a production $p : (L_T \stackrel{l}{\leftarrow} K_T \stackrel{r}{\rightarrow} \mathcal{G}_{T_p}) \in P$ with $\mathcal{G}_{T_p} = (T_p, G_{T_p}, P_p)$, and a matching $m : L_T \to G_{in}$, we proceed as follows (see Figure 4):

- We check that m and $l : K_T \to L_T$ satisfy the gluing conditions.
- We build the pushout complement of $\langle m, l \rangle$, obtaining a T-typed graph D with morphisms $k : K_T \to D$ and $b : D \to G_{in}$.

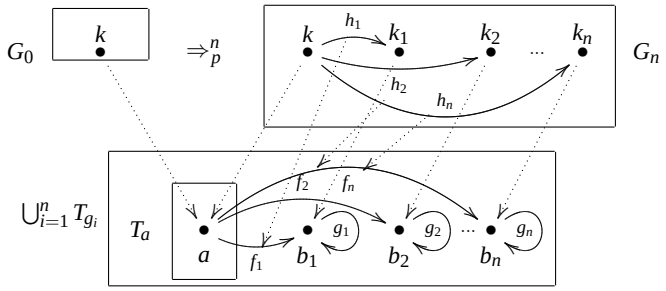


Fig. 5. A derivation where p is applied n times.

- We build the standard type graph T_{T,T_p} associated with $\sigma = id_T : T \to T$ and $\iota_p : T \hookrightarrow T_p$. Note that $\iota_{T,T_p} = \rho_{T,T_p} \circ \iota_p$. Fresh items of the underlying graph produced by the application of p must be typed over T_{T,T_p}.
- We build the retyped graphs $D' = \iota_{T,T_p} \cdot D$ and $K' = \iota_{T,T_p} \cdot K_T$ and take the morphism $k' : K' \to D'$ induced by k.
- We build the retyped DynGG $\mathcal{G}'_p = \rho_{T,T_p} \cdot \mathcal{G}_{T_p} = (T_{T,T_p}, G'_p, P')$ with $G'_p = \rho_{T,T_p} \cdot G_{T_p}$ and $P' = \rho_{T,T_p} \cdot P_p$ and take the morphism $r' : K' \to G'_p$ induced by $r : \iota_p \cdot K_T \to G_{T_p}$.
- We take the pushout of k' and r', resulting in a T_{T,T_p}-typed graph H with morphisms $d : D' \to H$ and $h : G'_p \to H$.
- Finally, we build the DynGG $\mathcal{G}' = (T_{T,T_p}, H, P' \cup \iota_{T,T_p} \cdot P)$.

When all the above is applicable, we say that there is a *direct dynamic derivation* α from $\mathcal{G}$ to $\mathcal{G}'$ using p (based on m) and write $\alpha : \mathcal{G} \Rightarrow_p \mathcal{G}'$. A *dynamic derivation* is a sequence of direct dynamic derivations starting from the initial graph

Example 3.1. Let us consider the T_a-typed grammar $\mathcal{G}_a$ presented in the Introduction (see productions p and q in Figures 1 and 2). The configuration after n applications of p is shown in Figure 5: the type graph has evolved from T_a to $\bigcup_{i=1}^n T_{g_i}$ (with $T_a = \bigcap_{i=1}^n T_{g_i}$) and there are $n+1$ available productions $p', q_1, ..., q_n$ at the top level that are suitable retyped instances of p and q. However, it is not possible to find a valid matching for any q_i, while there is (always) exactly one valid matching for the application of p.

3.1 About Shallow Graph Grammars

In the case of shallow grammars, the definition of derivation boils down to classic DPO derivation. This can be easily proved by noting that the fresh retyping leads to $T_{T,T_p} = T$ (i.e., the typing is vacuous) and that $P' = \emptyset$ (by definition of shallow grammars).

Proposition 3.1. *DynGGs are a conservative extension of GGs.*

The proof takes any T-typed graph grammar $\mathcal{G} = (T, G_{in}, P)$ and constructs the corresponding T-typed shallow graph grammar $\mathbf{Sh}(\mathcal{G})$. By what said above, it is then immediate to prove that $\delta : G \Rightarrow_p G'$ iff $\delta : \mathbf{Sh}(T, G, P) \Rightarrow_p \mathbf{Sh}(T, G', P)$.

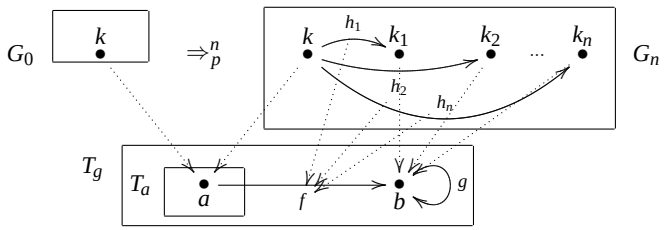


Fig. 6. A flattened derivation where p is applied n times.

Since the flattening $\mathbf{F}(\mathcal{G}) = (\mathbf{T}(\mathcal{G}), \mathfrak{t}_T \cdot G_{in}, \mathbf{P}(\mathcal{G}))$ of a DynGG $\mathcal{G} = (T, G_{in}, P)$ is also shallow, an obvious question is "how are the behaviours of $\mathbf{F}(\mathcal{G})$ and $\mathcal{G}$ related?"

Proposition 3.2. *Let $\mathcal{G}_0$ be a DynGG, then $\{\delta_i : \mathcal{G}_{i-1} \Rightarrow_{p_i} \mathcal{G}_i\}_{i\in\{1,..,n\}}$ implies $\{\delta'_i : \mathcal{G}'_{i-1} \Rightarrow_{\mathsf{flat}(p'_i)} \mathcal{G}'_i\}_{i\in\{1,..,n\}}$. where $\mathcal{G}'_0 = \mathbf{F}(\mathcal{G}_0)$ and each p_i is instance of $p'_i \in \mathbb{P}(\mathcal{G})$.*

The proof shows that there is a standard mapping from the dynamically evolving type graph of any $\mathcal{G}$ to the static representative $\mathbf{T}(\mathcal{G})$ and also a mapping from productions dynamically originated by $\mathcal{G}$ to the productions of $\mathbf{F}(\mathcal{G})$. In particular, any two different freshly-generated instances q', q'' of the same production q are mapped to $\mathsf{flat}(q)$. We remind that, contrary to $\mathcal{G}$, all productions in $\mathbf{F}(\mathcal{G})$ are always available and cannot change over time. Thus, any valid match for the dynamic graph remains valid in its flattening (via the retyping) and any direct derivation using the instance p_i of $p'_i \in \mathbb{P}(\mathcal{G})$ can be simulated using $\mathsf{flat}(p'_i) \in \mathbf{P}(\mathcal{G})$.

The counterexample below shows that $\mathbf{F}(\mathcal{G})$ has possibly more derivations than $\mathcal{G}$.

Example 3.2. Let us take the flattening $\mathbf{F}(\mathcal{G})$ of the T_a-typed DynGG $\mathcal{G}$ in Example 3.1. The configuration after n applications of p is in Figure 6. Note that the type graph remains $T_g = \mathbf{T}(\mathcal{G})$ and that there are only two available productions p, q at any time. Compare the situation with that in Figure 5: in the flattened version it is now possible to apply q to any pair of (distinct) arcs h_i, h_j!

4 Case studies: Dynamic nets and Join calculus

Dynamic nets [1,2] are an extension of Petri nets where firings can add fresh places and transitions. In this sense, any DynGG $\mathcal{G} = \langle T, G_{in}, P\rangle$ where $\mathbf{T}(\mathcal{G})$ is a discrete graph can be seen as a dynamic net. Nevertheless, not all dynamic nets can be represented by DynGGs over discrete type graphs because tokens may be coloured with the names of places in the net, and transitions may use such colours to designate places where to spawn new tokens. Since dynamic nets are in one-to-one correspondence with processes of the Join calculus [2], we present the encoding of the latter. Let $\mathcal{N}$ be an infinite set of names ranged by $u, v, x, y, z, \ldots$. The syntax of Join is given by the grammar

$$P ::= 0 \mid x\langle y\rangle \mid \mathbf{def}\ D\ \mathbf{in}\ P \mid P|Q \qquad D ::= J \triangleright P \mid D \wedge D \qquad J ::= x\langle y\rangle \mid J|J$$

The occurrences of x and u in $x\langle u\rangle$ are *free*. Differently, x and y occur bound in $P =$ **def** $x\langle u\rangle|y\langle v\rangle \triangleright P_1$ **in** P_2, while u and v occur bound in $D = x\langle u\rangle|y\langle v\rangle \triangleright P_1$. The sets of free and bound names of P are written respectively $fn(P)$ and $bn(P)$. Moreover, x and y are the defined names of D (written $dn(D)$).

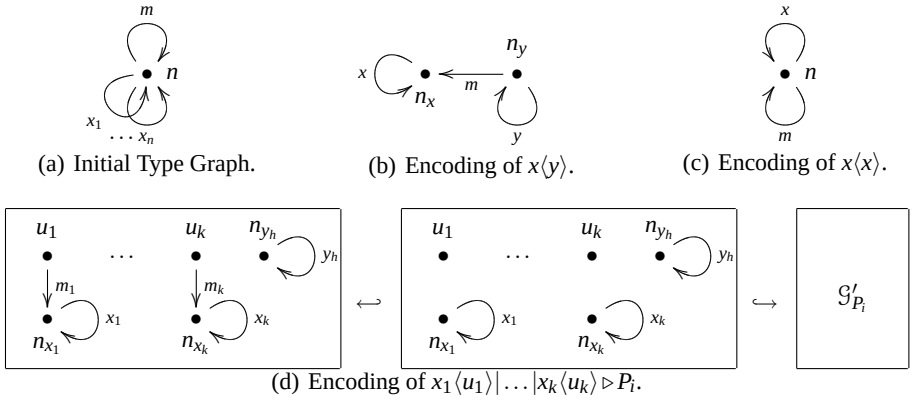
The semantics of the Join calculus relies on the *reflexive chemical abstract machine* model [9]. In this model a solution is roughly a multiset of active definitions $J \triangleright P$ and messages $x\langle u\rangle$ (separated by comma). Moves are distinguished between *structural* $\rightleftharpoons$, which heat or cool processes, and reductions $\rightarrow$, which are the basic computational steps. The multiset rewriting rules for Join are as follows:

$$0 \rightleftharpoons \qquad P \mid Q \rightleftharpoons P,Q \qquad D \wedge E \rightleftharpoons D,E$$
$$\textbf{def } D \textbf{ in } P \rightleftharpoons D\sigma_{dn(D)}, P\sigma_{dn(D)} \quad (range(\sigma_{dn(D)}) \text{ globally fresh})$$
$$J \triangleright P, J\sigma \rightarrow J \triangleright P, P\sigma$$

Structural moves allow for the rearrangement of terms inside a solution. Note that the term denoting a process with local definitions can be represented by two terms (one for the definitions and other for the process) only when the locally defined ports are renamed by fresh names (this rule stands for the dynamic generation of new names). A reduction can take place when the solution contains a rule $J \triangleright P$ and an instance $J\sigma$ of the Join pattern J: when such a match is found, $J\sigma$ is replaced by $P\sigma$. We write $P \mapsto P'$ for $P \rightleftharpoons^* Q \rightarrow Q' \rightleftharpoons^* P'$.

Join processes as DynGGs. For simplicity we assume definitions not to share names. Any process P is encoded as a DynGG $\mathcal{G}_P = \langle T_P, G_{in}, Q\rangle$. Generally speaking, a channel x will be encoded as node n but the fact that the channel is named x is denoted by an arc $x : n \rightarrow n$. A message $x\langle y\rangle$ is represented with the arc $m : n_1 \rightarrow n_2$, where n_1 corresponds to x and n_2 to y. Any firing rule $J \triangleright P$ will be encoded as a production. More formally, the initial type graph T_P is shown in Figure 7(a), where $fn(P) \cup bn(P) = \{x_1, \ldots, x_n\}$. T_P has a unique node n standing for channels, one arc m for denoting messages, and one arc x_i for any free or bound name of P. We call the *context* of P the T_P-typed graph C_P with one node n_{x_i} and one arc $x_i : n_{x_i} \rightarrow n_{x_i}$ for each $x_i \in fn(P) \cup bn(P)$. Then, the initial graph G_{in} and the set of productions Q are inductively defined as follows:

- $P = 0$. $G_{in} = C_P$ is the empty graph and $Q = \emptyset$.
- $P = x\langle y\rangle$. If $x \neq y$, then $G_{in} = C_P \cup \{m : n_x \rightarrow n_y\}$ is the graph shown in Figure 7(b), with the typing morphism mapping both nodes to n and being the identity on arcs. Otherwise, $G_{in} = C_P \cup \{m : n_x \rightarrow n_x\}$ is as in Figure 7(c). In both cases $Q = \emptyset$
- $P =$ **def** $J_1 \triangleright P_1 \wedge \ldots \wedge J_n \triangleright P_n$ **in** P'. Let $\mathcal{G}_{P'} = \langle T_{P'}, G'_{in}, Q'\rangle$ be the encoding of P', then $\mathcal{G}_P = \langle T_P, C_P \cup G'_{in}, Q' \cup \bigcup_{1\leq i\leq n}\{p_i\}\rangle$, where $T_P \supseteq T_{P'}$ and p_i encodes $J_i \triangleright P_i$. Assuming $J_i = x_1\langle u_1\rangle|\ldots|x_k\langle u_k\rangle$, then p_i is shown in Figure 7(d), where $\mathcal{G}'_{P_i}$ is the extension of $\mathcal{G}_{P_i} = \langle T_i, G_{in_i}, Q_i\rangle$ over the type graph $T_i \cup T_P$ and whose initial graph is the union of G_{in_i} with the items preserved by the production. The self-loop arcs naming the nodes u_i are not present in p_i because the identities of formal parameters are not known a priori and they will be provided by valid matchings. Moreover, the left-hand-side and the interface contain a node n_{y_h} and an arc y_h for any free name y_h of P_i not in $\{x_1, \ldots, x_k, u_1, \ldots, u_k\}$. In this way the context of the initial graph of $\mathcal{G}_{P_i}$ is bound to the names of the left-hand-side of the production.



(a) Initial Type Graph. (b) Encoding of $x\langle y\rangle$. (c) Encoding of $x\langle x\rangle$.

(d) Encoding of $x_1\langle u_1\rangle|\ldots|x_k\langle u_k\rangle \triangleright P_i$.

Fig. 7. Join processes as Dynamic Graph Grammars.

– $P = P_1|P_2$. Let $\mathcal{G}_{P_1} = \langle T_1, G_{in_1}, Q_1\rangle$ and $\mathcal{G}_{P_2} = \langle T_2, G_{in_2}, Q_2\rangle$ be the encoding of P_1 and P_2, then the initial graph is the pushout object of the span $C_P \cup G_{in_1} \hookleftarrow C_P \hookrightarrow C_P \cup G_{in_2}$, and Q is the union of Q_1 and Q_2 (upon production retyping over T_P).

Example 4.1. Let $P = \mathbf{def}\ x\langle u\rangle \triangleright (\mathbf{def}\ y\langle v\rangle \triangleright v\langle y\rangle\ \mathbf{in}\ y\langle u\rangle|x\langle y\rangle)\ \mathbf{in}\ x\langle z\rangle$. The corresponding grammar is $\mathcal{G}_P = \langle T_P, G_{in}, \{p\}\rangle$, where T_P and G_{in} are shown in Figure 8(a). The unique top-level production p is in Figure 8(b) (for space reasons we omit the representation of the typing). The right-hand-side of p is typed over the graph T' that adds two fresh arc types $y : n \to n$ and $u : n \to n$ to T_P. Production p describes the consumption of a message sent to the channel x regardless of the name contained by the message (note that the particular name of the port n_u is not fixed by the production). When fired, p generates two fresh types $y' : n \to n$ and $u' : n \to n$ and modifies the underlying graph by removing m_1 and by adding (*i*) a new node $n_{y'}$, (*ii*) a new arc of the fresh type y', (*iii*) a new arc of the fresh type u' that works as an alias for the actual name of the actual parameter n_u, and (*iv*) two new messages m_2 and m_3. Moreover, p spawns a new production q (Figure 8(c)), which handles the messages sent to the fresh port y'.

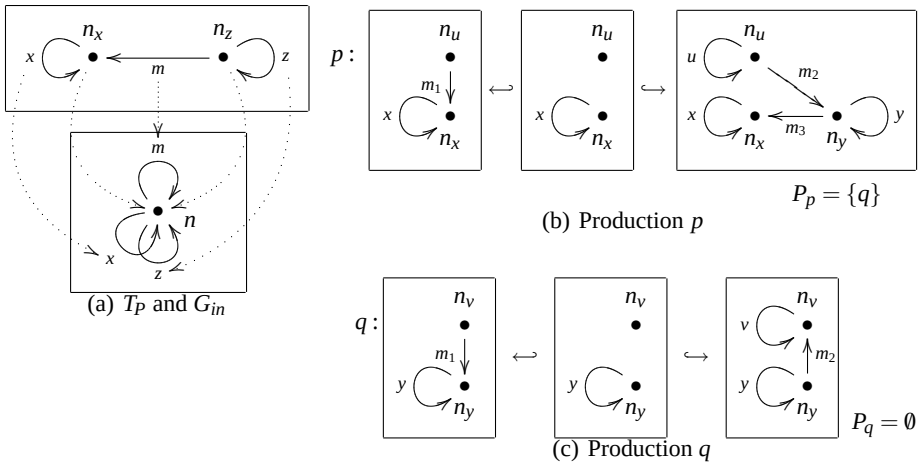
The encoding of Join processes establishes a tight correspondence between derivations in the two frameworks. The following results hold up to aliasing of names (i.e., by removing aliasing from grammars).

Proposition 4.1. *For any Join process P we have:*

– *If $P \mapsto P'$ using $J_i \triangleright P_i$, then $\exists Q$ s.t. $\mathcal{G}_P \Rightarrow_{p_i} \mathcal{G}_Q$ and $Q \rightleftharpoons^* P'$;*
– *If $\mathcal{G}_P \Rightarrow_{p_i} \mathcal{G}'$, then $\exists P'$ s.t. $P \mapsto P'$ using $J_i \triangleright P_i$ and $\mathcal{G}' = \mathcal{G}_{P'}$.*

5 Encoding Dynamic Graph Grammars as Graph Grammars

In this section we show that DynGGs can be encoded back in GGs. The encoding of a Dynamic Graph Grammar $\mathcal{G}$ relies on the definition of a unique type graph expressive



(a) T_P and G_{in} (b) Production p (c) Production q

Fig. 8. A Join process as a DynGG.

enough for distinguishing all the types generated dynamically by $\mathcal{G}$. As a first step, we show how to describe a chain of types $\mathcal{T}$ ordered by inclusions $T_1 \subset T_2 \subset \ldots \subset T_n$ with a unique type graph $\{[T_n]\}_{\mathcal{T}}$, called the *refined type graph*. Informally, any item (i.e., node or arc) of T_n is mapped to a node in $\{[T_n]\}_{\mathcal{T}}$. Every graph T_i in the chain $\mathcal{T}$ is also represented in $\{[T_n]\}_{\mathcal{T}}$ by a node $\mathtt{n}_{T_i}$. Moreover, any node w corresponding to an item k of T_n has an arc $\mathtt{t}_w$ to the node $\mathtt{n}_{T_i}$ if T_i is the minimal type in $\mathcal{T}$ that includes k. We call T_i the *type of* k in $\mathcal{T}$. Formally, for any $k \in T_n$, the type of k is $\mathcal{T}(k) = T_i$ if $k \in T_i \backslash T_{i-1}$.

Definition 5.1 (Refined type graph). *Given a type graph T and a chain of types $\mathcal{T} = T_1 \subset \ldots \subset T_n$, with $T_n = T$, the* refined type graph *is $\{[T]\}_{\mathcal{T}} = \langle N_R, E_R, s_R, t_R \rangle$, where:*

- *$N_R = N_T \cup E_T \cup \{\mathtt{n}_{T_i} | T_i \in \mathcal{T}\}$ where $\mathtt{n}_{T_i}$ are fresh names, i.e., the nodes of $\{[T]\}_{\mathcal{T}}$ are the nodes and arcs of T plus one extra node for any type in $\mathcal{T}$;*
- *$E_R = \{e_0, e_1 | e \in E_T\} \cup \{\mathtt{t}_w | w \in N_T \cup E_T\} \cup \{\mathtt{s}_{i,i+1} | 0 < 1 < n-1\}$ where all edge names are fresh. Source and target functions are defined s.t. the following holds:*
  - *$e_0 : s_T(e) \to e$ and $e_1 : e \to t_T(e)$, i.e., e_0 connects the node $e \in N_R$ to its original source in T while e_1 connects e to its target;*
  - *$\mathtt{t}_w : w \to \mathtt{n}_{\mathcal{T}(w)}$, i.e., $\mathtt{t}_w$ connects w to the node representing its type;*
  - *$\mathtt{s}_{i,i+1} : \mathtt{n}_{T_i} \to \mathtt{n}_{T_{i+1}}$ denotes the inclusion of types $T_i \subset T_{i+1}$.*

Example 5.1. Consider the type graph T_g depicted in the bottom part of the Figure 9(a). The refined type graph for the chain $T_a = \{\mathtt{a}\} \subset T_g = T_a \cup \{f, b, g\}$ is shown at the bottom of Figure 9(b). The original arc f (resp. g) of T_g is represented by the homonymous node f (resp. g) and the pair of fresh arcs f_0 and f_1 (resp. g_0 and g_1). The types T_a and T_g are represented by the fresh nodes $\mathtt{n}_{T_a}$ and $\mathtt{n}_{T_g}$, while the inclusion relation $T_a \subset T_g$ is denoted by the arc $\mathtt{s}_{1,2}$. Finally, for any item w, $\mathtt{t}_w$ connects w to its type node.

Definition 5.2 (Refined T-Typed Graph). *Given a T-typed graph $G = \langle |G|, \tau_G \rangle$ and a chain $\mathcal{T} = T_1 \subset \ldots \subset T_n = T$, the $\{[T]\}_{\mathcal{T}}$-typed graph $\{[G]\}_{\mathcal{T}} = \langle |H|, \tau_H \rangle$ is defined as:*

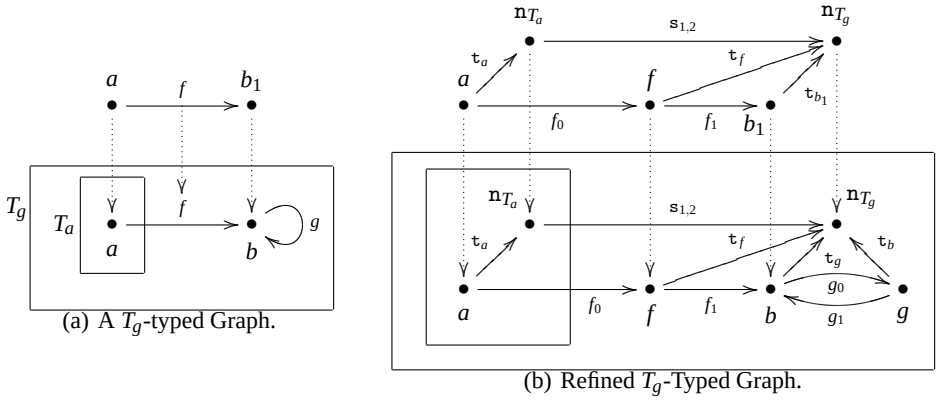


Fig. 9. A refined T-Typed Graph.

- $N_H = N_G \cup E_G \cup \{\mathtt{n}_{T_i} | T_i \in \mathcal{T}\}$, *i.e., N_H has all items of G plus nodes denoting types;*
- $E_H = \{e_0, e_1 | e \in E_G\} \cup \{\mathtt{t}_w | w \in N_G \cup E_G\} \cup \{\mathtt{s}_{i,i+1} | 0 < 1 < n-1\}$, *where:*
  - $e_0 : s_T(e) \to e$ *and* $e_1 : e \to t_T(e)$*;*
  - $\mathtt{t}_w : w \to \mathtt{n}_{\mathcal{T}(\tau_G(w))}$, *i.e., $\mathtt{t}_w$ connects w to the node representing its type in $\mathcal{T}$, which is obtained by using the typing morphism $\tau_G(w)$;*
  - $\mathtt{s}_{i,i+1} : \mathtt{n}_{T_i} \to \mathtt{n}_{T_{i+1}}$, *for the inclusion of types.*
- *The typing morphism τ_H is defined as follows*

$$\tau_H(k) = \tau_G(k) \text{ if } k \in G \qquad \tau_H(\mathtt{n}_{T_i}) = \mathtt{n}_{T_i}$$
$$\tau_H(e_i) = \tau_G(e)_i \qquad \tau_H(\mathtt{t}_w) = \mathtt{t}_{\tau_G(w)} \qquad \tau_H(\mathtt{s}_{i,i+1}) = \mathtt{s}_{i,i+1}$$

Example 5.2. Consider the T_g-typed graph G in Figure 9(a). Its refined version for the chain $T_a = \{\mathtt{a}\} \subset T_g = T_a \cup \{f, b, g\}$ is shown in Figure 9(b) (we omit the representation of the obvious typing of arcs).

We refer to the nodes $\mathtt{n}_{T_i}$ and the arcs $\mathtt{t}_w$ and $\mathtt{s}_{i,i+1}$ of a refined type graph (resp., a refined T-typed graph) as the *location of the type graph* (resp., *location of the graph*).

Definition 5.3 (Refined T-Typed DynGG). *Let $\mathcal{G} = (T, G_{in}, P)$ be a DynGG, and $\mathcal{T} = T_1 \subset \ldots \subset T_n$, with $T_n = T$ be a chain of types. Then, the refined version of $\mathcal{G}$ is defined as $\mathcal{G}_{\mathcal{T}} = (\{[T]\}_{\mathcal{T}}, \{[G_{in}]\}_{\mathcal{T}}, \{[P]\}_{\mathcal{T}})$, where $\{[P]\}_{\mathcal{T}} = \{\{[p]\}_{\mathcal{T}} | p \in P\}$ is obtained by encoding any production $p : (L \stackrel{l}{\leftarrow} K \stackrel{r}{\rightarrow} (T', G'_{in}, P'))$ in P as follows:*

$$\{[p]\}_{\mathcal{T}} : (\{[L]\}_{\mathcal{T}} \stackrel{l'}{\leftarrow} \{[K]\}_{\mathcal{T}} \stackrel{r'}{\rightarrow} \{[(T', G'_{in}, P')]\}_{\mathcal{T} \subset T'})$$

where morphisms l' and r' are the obvious extensions of l and r with the identity over the location of the graph.

Example 5.3. Consider the production p in Figure 1. Its refined version is in Figure 10. The type graphs are the refined versions of the original type graphs, while the left-hand-side, the interface, and the right-hand-side are the refined version of the original ones.

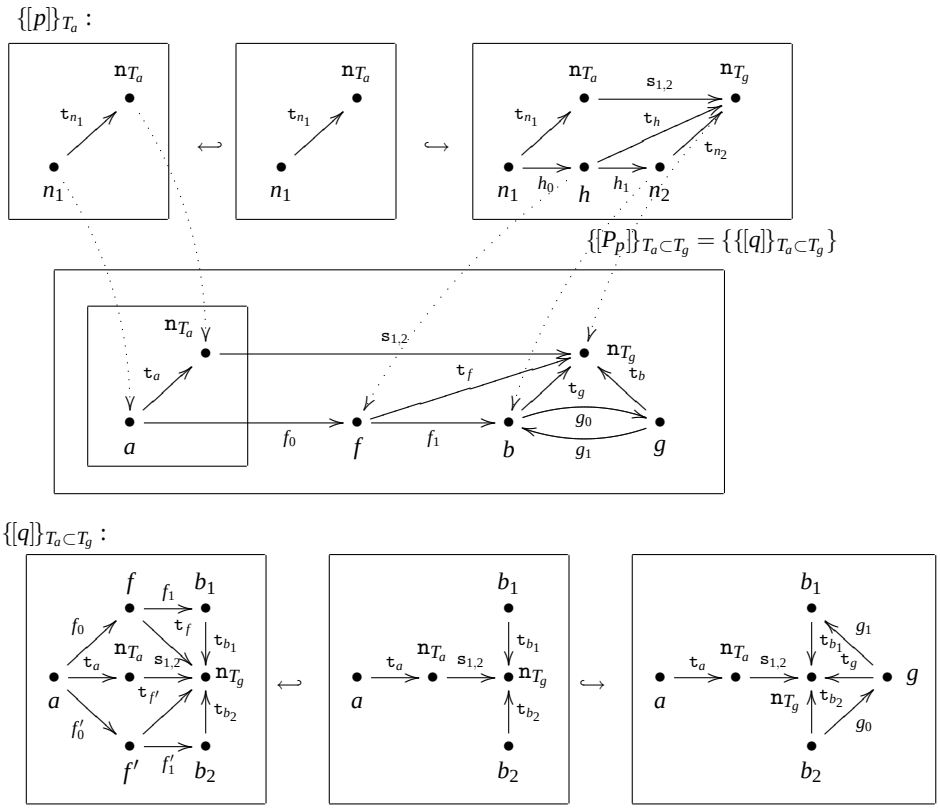


Fig. 10. A refined production p.

In particular, the left-hand-side is typed over the refined version of T_a, while the right-hand-side grammar is typed over the refined version of T_g. Moreover, the production $\{[q]\}_{T_a \subset T_g}$ created by the reduction corresponds to the refined versions of the original q (for clarity we do not draw the typing morphism, which is the obvious one).

The refined version of a grammar $\mathcal{G}$ recreates the static tree of types $\mathbf{T}(\mathcal{G})$. In fact, any production $p : (L_T \xleftarrow{l'} K_T \xrightarrow{r'} \mathcal{G}_{T'})$ is encoded by considering the path $\mathcal{T}$ of $\mathbf{T}(\mathcal{G})$ starting from the root of $\mathbf{T}(\mathcal{G})$ to T. Moreover, since previous definitions can be straightforwardly extended to consider the whole tree instead of a path, we will use $\{[\_]\}_{\mathcal{T}}$ to denote also the refined versions obtained by considering the tree of types $\mathcal{T}$. Given any tree $\mathcal{T}$ describing type inclusions, the tree $\mathcal{T}' = \mathcal{T}, T \hookrightarrow T'$ stands for the tree $\mathcal{T}$ with the addition of the type T' as a child of T (if T' is already in the tree, then $\mathcal{T}' = \mathcal{T}$).

Remark 5.1. For simplicity, we assume the name of any production to be decorated with the types of its left and right-hand-sides, i.e., $p_{T \hookrightarrow T'} : (L_T \xleftarrow{l} K_T \xrightarrow{r} \mathcal{G}_{T'})$.

The result below shows that a refined grammar behaves like the original one.

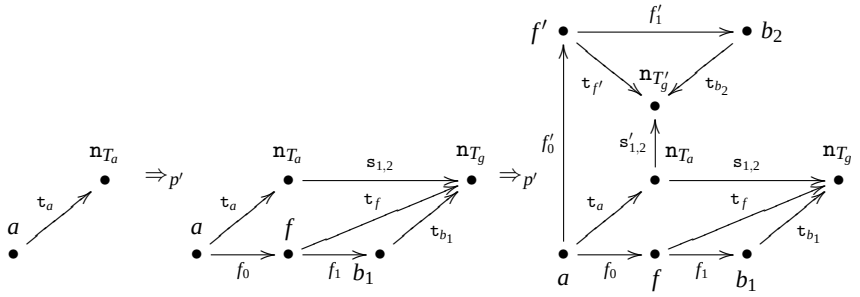


Fig. 11. A flattened, refined derivation.

Lemma 5.1. *Let $\mathcal{G}_0 = (T_0, H_0, P_0)$ and $\mathcal{G}_n = (T_n, H_n, P_n)$ be DynGGs, then*

$$\{\delta_i : \mathcal{G}_i \Rightarrow_{p_i, T_{p_i} \hookrightarrow T'_{p_i}} \mathcal{G}_{i+1}\}_{i\in\{0,..,n-1\}} \quad \textit{iff} \quad \{\delta'_i : \{[\mathcal{G}_i]\}_{\mathcal{T}_i} \Rightarrow_{p_i} \{[\mathcal{G}_{i+1}]\}_{\mathcal{T}_{i+1}}\}_{i\in\{0,..,n-1\}}$$

where $\mathcal{T}_0 = T_0$ and $\mathcal{T}_{i+1} = \mathcal{T}_i, T_{p_i} \hookrightarrow T'_{p_i}$ for $1 \le i \le n$.

Proof (Sketch). Consider $\mathcal{G}_i \Rightarrow_{p_{T\hookrightarrow T'}} \mathcal{G}_{i+1}$ and $p_{T\hookrightarrow T'} : (L_T \xleftarrow{l} K_T \xrightarrow{r} \mathcal{G}_{T'})$. Then, the derivation $\mathcal{G}_i \Rightarrow_{T\hookrightarrow T'} \mathcal{G}_j$ is analogous to that one in Figure 4. By construction of $\{[\mathcal{G}_i]\}_{\mathcal{T}_i}$, there exists $m : L_T \to G_{in}$ iff there exists $m' : \{[L_T]\}_{\mathcal{T}_i} \to \{[G_{in}]\}_{\mathcal{T}_i}$ in the refined grammar. Since $\{[K_T]\}_{\mathcal{T}_i}$ preserves the "same" elements (up-to suitable encoding) as K_T plus the location of the graph, then D' obtained as the pushout complement of $\langle m', l'\rangle$ coincides with $\{[D]\}_{\mathcal{T}}$. Since $\{[R_T]\}_{\mathcal{T}_i}$ preserves the location of already existing items and generates a new node $\mathtt{n}_{T'}$ and a new arc $\mathtt{s} : \mathtt{n}_T \to \mathtt{n}_{T'}$ for typing fresh items, then H' coincides with $\{[H]\}_{\mathcal{T}_i, T\hookrightarrow T'}$. The correspondence among fresh productions is straightforward.

Definition 5.4 (Encoding). *Let $\mathcal{G} = \langle T, G_{in}, P\rangle$ be a DynGG. Then, the equivalent graph grammar $\{[\mathcal{G}]\}$ is defined as $\{[\mathcal{G}]\} = \mathbf{F}(\{[\mathcal{G}]\}_T)$.*

Example 5.4. Consider the DynGG $\mathcal{G} = \langle T_a, G_{in}, \{p\}\rangle$ with T_a and p as in Figures 1 and 2 and its encoding $\{[\mathcal{G}]\} = \langle T', \{[G_{in}]\}_{T_a}, \{p', q'\}\rangle$, where T' is the type graph in Figure 10, and p' and q' are analogous to $\{[p]\}_{T_a}$ and $\{[q]\}_{T_a \subset T_g}$ in Figure 10. Figure 11 shows a derivation that applies twice the rule p' over the initial graph consisting of a unique node typed a. Although the final graph contains two arrows of type f with same source of type a, there is not a matching for q', since the left-hand-side of q' requires the targets of the two arrows to have the same location. Hence, the encoding does not confuse different instantiations of the same type, as formalised by the following result.

Theorem 5.1 (Correspondence). *Let $\mathcal{G}_0$ be a dynamic graph grammar, then*

$$\{\delta_i : \mathcal{G}_{i-1} \Rightarrow_{p_i} \mathcal{G}_i\}_{i\in\{0,..,n\}} \quad \textit{iff} \quad \{\delta'_i : \mathcal{G}'_{i-1} \Rightarrow_{\{[p_i]\}} \mathcal{G}'_i\}_{i\in\{0,..,n\}}$$

where $\mathcal{G}'_0 = \{[\mathcal{G}_0]\}$ and $\{[p]\}$ is the encoding of the rule p in $\mathbf{F}(\mathcal{G}_0)$.

Proof. $\Rightarrow$) Follows immediately by Lemma 5.1 and Proposition 3.2. $\Leftarrow$) It remains to prove that the matchings on the encoded version are the same as those of the original one. In fact, the encoding of a rule assures the location of any graphs to identify items with the same type while differentiating items with distinct types.

6 Concluding Remarks

We have proposed the original framework of Dynamic Graph Grammars, as a conservative extension of Graph Grammars that offers a convenient level of abstraction for modeling reflexive systems. Our main result proves that Dynamic Graph Grammars can be simulated by ordinary Graph Grammars, though a non-trivial encoding is necessary.

When compared to the vast literature of theoretical foundations and applications of graph transformation systems, our investigation on reflexive productions is still preliminary under many aspects. A fully extensive development and assessment is therefore a very ambitious programme, along which we foresee several promising directions: (1) to express suitable notion of independent derivations, parallelism, process semantics, unfolding semantics and event structure semantics so to fully develop a true concurrent semantics of DynGGs; (2) to show that concurrency is preserved by our encoding of DynGGs in GGs; (3) to consider other flavours of dynamic productions, like the SPO [12,7]; (4) to exploit the encoding in § 5 to reuse verification tools developed for GGs for systems modeled using DynGGs.

Acknoweldgement. The authors want to thank Ivan Lanese for many helpful discussions on the encoding of DynGGs back to GGs.

References

1. A. Asperti and N. Busi. Mobile Petri nets. Technical Report UBLCS 96-10, Computer Science Department, University of Bologna, 1996.
2. M. Buscemi and V. Sassone. High-level Petri nets as type theories in the Join calculus. *Proc. of FoSSaCS'01*, *Lect. Notes in Comput. Sci.* 2030, pp. 104–120. Springer, 2001.
3. A. Corradini, P. Degano, and U. Montanari. Specifying highly concurrent data structure manipulation. *Proc. of Computing'85: A Broad Perspective of Current Developments*. 1985.
4. A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. Algebraic approaches to graph transformation I: Basic concepts and double pushout approach. In [13].
5. A. Corradini, U. Montanari, and F. Rossi. Graph processes. *Fund. Inf.*, 26:241–265, 1996.
6. F. Drewes, H.-J. Kreowski, and A. Habel. Hyperedge replacement graph grammars. In [13].
7. H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini. Algebraic approaches to graph transformation II: Single pushout approach and comparison with double pushout approach. In [13].
8. J. Engelfriet and G. Rozenberg. Node replacement graph grammars. In [13].
9. C. Fournet and G. Gonthier. The reflexive chemical abstract machine and the Join calculus. *Proc. of POPL'96*, pp. 372–385. ACM Press, 1996.
10. I. Lanese. *Synchronization Strategies for Global Computing Models*. PhD thesis, Department of Computer Science, University of Pisa, 2006.
11. D. Hirsch. *Graph Transformation Models for Software Architecture Styles*. PhD thesis, Departamento de Computación, Universidad de Buenos Aires, 2003.
12. M. Löwe. Algebraic approach to single-pushout graph transformation. *Theoret. Comput. Sci.*, 109:181–224, 1993.
13. G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation. Vol. 1: Foundations*. World Scientific, 1997.