Mapping Fusion and Synchronized Hyperedge Replacement into Logic Programming∗

IVAN LANESE and UGO MONTANARI
Dipartimento di Informatica, Università di Pisa
(*e-mail:* {lanese,ugo}@di.unipi.it)

submitted 23 December 2003; revised ?; accepted ?

Abstract

In this paper we compare three different formalisms that can be used in the area of models for distributed, concurrent and mobile systems. In particular we analyze the relationships between a process calculus, the *Fusion Calculus*, graph transformations in the *Synchronized Hyperedge Replacement* with Hoare synchronization (HSHR) approach and *logic programming*. We present a translation from Fusion Calculus into HSHR (whereas Fusion Calculus uses Milner synchronization) and prove a correspondence between the reduction semantics of Fusion Calculus and HSHR transitions. We also present a mapping from HSHR into a synchronized version of logic programming and prove that there is a full correspondence between the two formalisms. The resulting mapping from Fusion Calculus to logic programming is interesting since it shows the tight analogies between the two formalisms, in particular for handling name generation and mobility. The intermediate step in terms of HSHR is convenient since graph transformations allow for multiple, remote synchronizations, as required by Fusion Calculus semantics.

KEYWORDS: Fusion Calculus, graph transformation, Synchronized Hyperedge Replacement, logic programming, mobility

1 Introduction

In this paper we compare different formalisms that can be used to specify and model systems which are distributed, concurrent and mobile, as those that are usually found in the global computing area.
Global computing is becoming very important because of the great development of networks which are deployed on huge areas, first of all internet, but also other kinds of networks such as networks for wireless communications. In order to build and program these networks one needs to deal with issues such as reconfigurability, synchronization and transactions at a suitable level of abstraction. Thus powerful formal models and tools are needed. Until now no model has been able to emerge as the standard one for this kind of systems, but there are a lot of approaches with different merits and drawbacks.

∗ Work supported in part by the European IST-FET Global Computing project PROFUNDIS.

An important approach is based on process-calculi, like Milner's CCS and Hoare's CSP. These two calculi deal with communication and synchronization in a simple way, but they lack the concept of mobility. An important successor of CCS, the π-calculus (Milner et al. 1992), allows the study of a wide range of mobility problems in a simple mathematical framework. We are mainly interested in the *Fusion Calculus* (Parrow and Victor 1998; Victor 1998; Gardner and Wischik 2000; Gardner and Wischik 2004), which is an evolution of π-calculus. The interesting aspect of this calculus is that it has been obtained by simplifying and making more symmetric the π-calculus.

One of the known limitations of process-calculi when applied to distributed systems is that they lack an intuitive representation because they are equipped with an *interleaving* semantics and they use the same constructions for representing both the agents and their configurations. An approach that solves this kind of problems is based on *graph transformations* (Ehrig et al. 1999). In this case the structure of the system is explicitly represented by a graph which offers both a clean mathematical semantics and a suggestive representation. In particular we represent computational entities such as processes or hosts with hyperedges (namely edges with possibly more than two connections) and channels between them with shared nodes. As far as the dynamic aspect is concerned, we use *Synchronized Hyperedge Replacement* with *Hoare synchronization* (HSHR) (Degano and Montanari 1987). This approach uses productions to specify the behaviour of single hyperedges, which are synchronized by exposing actions on nodes. Actions exposed by different hyperedges on the same node must be compatible. In the case of Hoare synchronization all the edges must expose the same action (in the CSP style).
We use the extension of HSHR with *mobility* (Hirsch et al. 2000; Hirsch and Montanari 2001; König and Montanari 2001; Ferrari et al. 2001; Lanese 2002), that allows edges to expose node references together with actions, and nodes whose references are matched during synchronization are unified.

For us HSHR is a good step in the direction of *logic programming*. We consider logic programming as a formalism for modelling concurrent and distributed systems. This is a non-standard view of logic programming (see Bruni et al. 2001 for a presentation of our approach) which considers goals as processes whose evolution is defined by Horn clauses and whose interactions use variables as channels and are managed by the unification engine. In this framework we are not interested only in refutations, but in any partial computation that rewrites a goal into another.

In this paper we analyze the relationships between these three formalisms and we find tight analogies among them, like the same parallel composition operator and the use of unification for name mobility. However we also emphasize the differences between these models:

- the Fusion Calculus is interleaving and relies on *Milner synchronization* (in the CCS style);

- HSHR is inherently *concurrent* and uses Hoare synchronization;
- logic programming is concurrent, has a wide spectrum of possible controls which are based on the Hoare synchronization model, and also is equipped with a more complex data management.

We will show a mapping from Fusion Calculus to HSHR and prove a correspondance theorem. Note that HSHR is a good intermediate step between Fusion Calculus and logic programming since in HSHR hyperedges can perform multiple actions at each step (on different nodes), and this allows to build chains of synchronizations. This additional power is needed to model Milner synchronization (which requires synchronous, atomic routing capabilities) and fusions, which, while being triggered by two synchronizing processes, can affect remote agents. To simplify our treatment we consider only reduction semantics. The interleaving behaviour is imposed with an external condition on the allowed HSHR transitions.

Finally we present the connections between HSHR and logic programming. Since the logic programming paradigm allows for many computational strategies and is equipped with powerful data structures, we need to constraint it in order to have a close correspondence with HSHR. We obtain *Synchronized Logic Programming* (SLP), where edges are translated into predicates, nodes into variables and parallel composition into AND composition.

This translation was already presented in the MSc. thesis of the first author (Lanese 2002) and in Lanese and Montanari (2002). Fusion Calculus was mapped into SHR with Milner synchronization (a simpler task) in Lanese and Montanari (2003). The paper Lanese and Montanari (2002) also contains a mapping of Ambient calculus into HSHR.

Since logic programming is not only a theoretical framework, but also a well developed programming style, the connections between Fusion, HSHR and logic programming can be used for implementation purposes. SLP has been implemented in Lanese (2002) through meta-interpretation. Thus we can use translations from Fusion and HSHR to implement them.

In Section 2 we present the required background, in particular we introduce the Fusion Calculus (2.1), the algebraic representation of graphs and the HSHR (2.2), and logic programming (2.3). Section 3 is dedicated to the mapping from Fusion Calculus to HSHR. Section 4 analyzes the relationships between HSHR and logic programming, in particular we introduce SLP (4.1), we prove the correspondence between it and HSHR (4.2) and we give some hints on how to implement Fusion Calculus and HSHR using Prolog (4.3). Finally in Section 5 we present some conclusions and traces for future work.

2 Background

2.1 The Fusion Calculus

The Fusion Calculus (Parrow and Victor 1998; Victor 1998) is a calculus for modelling distributed and mobile systems which is based on the concepts of *fusion* and scope. It is an evolution of the π-calculus (Milner et al. 1992) and the interesting point is that it is obtained by simplifying the calculus. In fact the two action prefixes for input and output communication are symmetric, whereas in the π-calculus they are not and there is just one binding operator called scope, whereas the π-calculus has two (restriction and input). As shown in Parrow and Victor (1998), the π-calculus is syntactically a subcalculus of the Fusion Calculus (the key point is that the input of π-calculus is obtained using input and scope). In order to have these properties fusion actions have to be introduced. An asynchronous version of Fusion Calculus is described in Gardner and Wischik (2000), Gardner and Wischik (2004), where name fusions are handled explicitly as messages. Here we follow the approach by Parrow and Victor.

We now present in details the syntax and the reduction semantics of Fusion Calculus. In our work we deal with a subcalculus of the Fusion Calculus, which has no match and no mismatch operators, and has only guarded summation and recursion. In our discussion we distinguish between *sequential processes* (which have a guarded summation as topmost operator) and general processes.

We assume to have an infinite set $\mathcal{N}$ of names ranged over by $u, v, \ldots, z$. Names represent communication channels. We use ϕ to denote an equivalence relation on $\mathcal{N}$ which is represented in the syntax by a finite set of equalities. Function $\mathrm{n}(\phi)$ returns all names which are fused, i. e. those contained in an equivalence class of ϕ which is not a singleton.

Definition 1

The prefixes are defined by:

$$\begin{array}{rcll} \alpha & ::= & u\vec{x} & \text{(Input)} \\ & & \overline{u}\vec{x} & \text{(Output)} \\ & & \phi & \text{(Fusion)} \end{array}$$

Definition 2

The agents are defined by:

$$S ::= \sum_i \alpha_i.P_i \quad \text{(Guarded sum)}$$

$$\begin{array}{rcl} P ::= & 0 & \text{(Inaction)} \\ & S & \text{(Sequential Agent)} \\ & P_1|P_2 & \text{(Composition)} \\ & (x)P & \text{(Scope)} \\ & \mathrm{rec}\, X.P & \text{(Recursion)} \\ & X & \text{(Agent variable)} \end{array}$$

The scope restriction operator is a binder for names, thus x is bound in $(x)P$. Similarly rec is a binder for agent variables. We will only consider agents which are closed with respect to both names and agent variables and where in $\mathrm{rec}\, X.P$ each occurrence of X in P is within a sequential agent (guarded recursion).

Given an agent P, functions fn, bn and n compute the sets $\mathrm{fn}(P)$, $\mathrm{bn}(P)$ and $\mathrm{n}(P)$ of its free, bound and all names respectively.

Processes are agents considered up-to structural axioms defined as follows.

Definition 3 (*Structural congruence*)
The structural congruence $\equiv$ between agents is the least congruence satisfying the α-conversion law (both for names and for agent variables), the abelian monoid laws for summation and composition (associativity, commutativity and 0 as identity), the scope laws $(x)0 \equiv 0$, $(x)(y)P \equiv (y)(x)P$, the scope extension law $P|(z)Q \equiv (z)(P|Q)$ where $z \notin \mathrm{fn}(P)$ and the recursion law $\mathrm{rec}\, X.P \equiv P\{\mathrm{rec}\, X.P/X\}$.

Note that fn is also well-defined on processes.
In order to deal with fusions we need the following definition.

Definition 4 (*Substitutive effect*)
A substitutive effect of a fusion ϕ is any idempotent substitution $\sigma : \mathcal{N} \to \mathcal{N}$ having ϕ as its kernel. In other words $\sigma(x) = \sigma(y)$ iff $x\phi y$ and σ sends all members of each equivalence class of ϕ to one representative in the class[1].

The reduction semantics for Fusion Calculus is the least relation satisfying the following rules.

Definition 5 (*Reduction semantics for Fusion Calculus*)

$$(\vec{z})(R|(\cdots + u\vec{x}.P)|(\overline{u}\vec{y}.Q + \ldots)) \to (\vec{z})(R|P|Q)\sigma$$

where $|\vec{x}| = |\vec{y}|$ and σ is a substitutive effect of $\{\vec{x} = \vec{y}\}$.

$$(\vec{z})(R|(\cdots + \phi.P)) \to (\vec{z})(R|P)\sigma$$

where σ is a substitutive effect of ϕ.

$$\frac{P \equiv P', P' \to Q', Q' \equiv Q}{P \to Q}$$

[1] Essentially σ is a most general unifier of ϕ, when it is considered as a set of equations.

2.2 Synchronized Hyperedge Replacement

Synchronized Hyperedge Replacement (SHR) (Degano and Montanari 1987) is an approach to (hyper)graph transformations that defines *global transitions* using *local productions*. Productions define how a single hyperedge can be rewritten and the *conditions* that this rewriting imposes on adjacent nodes. Thus the global transition is obtained by applying in parallel different productions whose conditions are compatible. What exactly compatible means depends on which *synchronization model* we use. In this work we will use the Hoare synchronization model (HSHR), which requires that all the hyperedges connected to a node expose the same action on it. We use the extension of HSHR with *mobility* (Hirsch et al. 2000; Hirsch and Montanari 2001; König and Montanari 2001; Ferrari et al. 2001; Lanese 2002), that allows edges to expose node references together with actions, and nodes whose references are matched during synchronization are unified.

We will give a formal description of HSHR in terms of labelled transition system, but first of all we need an algebraic representation for hypergraphs.

A hyperedge, or simply an edge, is an atomic item with a label (from a ranked alphabet $LE = \{LE_n\}_{n=0,1,\ldots}$) and with as many ordered tentacles as the rank of its label. A set of nodes, together with a set of such edges, forms a hypergraph (or simply a graph) if each edge is connected, by its tentacles, to its attachment nodes. We will consider graphs up-to isomorphisms that preserve[2] nodes, labels of edges, and connections between edges and nodes.
Now, we present a definition of graphs as syntactic judgements, where nodes correspond to names and edges to basic terms of the form $L(x_1, \ldots, x_n)$, where x_i are arbitrary names and $L \in LE_n$.

Definition 6 (*Graphs as syntactic judgements*)
Let $\mathcal{N}$ be a fixed infinite set of names and LE a ranked alphabet of labels. A syntactic judgement (or simply a judgement) is of the form $\Gamma \vdash G$ where:

1. $\Gamma \subseteq \mathcal{N}$ is the (finite) set of nodes in the graph.
2. G is a term generated by the grammar
   $G ::= L(\vec{x}) \mid G|G \mid nil$
   where $\vec{x}$ is a vector of names and L is an edge label with $\mathrm{rank}(L) = |\vec{x}|$.

We denote with n the function that given a graph G returns the set $\mathrm{n}(G)$ of all the names in G. We use the notation Γ, x to denote the set obtained by adding x to Γ, assuming $x \notin \Gamma$. Similarly, we will write Γ_1, Γ_2 to state that the resulting set of names is the disjoint union of Γ_1 and Γ_2.

Definition 7 (*Structural congruence and well-formed judgements*)
The structural congruence $\equiv$ on terms G obeys the following axioms:

$$\text{(AG1)}\ (G_1|G_2)|G_3 \equiv G_1|(G_2|G_3)$$

[2] In our approach nodes usually represent free names, and they are preserved by isomorphisms.

$$\text{(AG2)}\ G_1|G_2 \equiv G_2|G_1$$

$$\text{(AG3)}\ G|nil \equiv G$$

The well-formed judgements for constructing graphs over LE and $\mathcal{N}$ are those generated by applying the following syntactic rules.

$$\text{(RG1)}\ \frac{x_i \in \mathcal{N}}{x_1, \ldots, x_n \vdash nil}$$

$$\text{(RG2)}\ \frac{L \in LE_m \qquad x_i \in \mathcal{N}, i = 1 \ldots n \qquad y_j \in \{x_1, \ldots, x_n\}, j = 1 \ldots m}{x_1, \ldots, x_n \vdash L(y_1, \ldots, y_m)}$$

$$\text{(RG3)}\ \frac{\Gamma \vdash G_1 \qquad \Gamma \vdash G_2}{\Gamma \vdash G_1|G_2}$$

Axioms (AG1),(AG2) and (AG3) define respectively the associativity, commutativity and identity over *nil* for operation $|$. Rule (RG1) creates a graph with no edges and n nodes and rule (RG2) creates a graph with n nodes and one edge labelled by L and with m tentacles (note that some tentacles can be attached to the same node). Rule (RG3) allows the composition of two graphs that share the same set of nodes. Note that if $\Gamma \vdash G$ is a well-formed judgement then $\mathrm{n}(G) \subseteq \Gamma$.

We can state the following correspondence theorem.

Theorem 1 (*Correspondence of graphs and judgements*)
Well-formed judgements up to structural axioms are isomorphic to graphs up to isomorphisms.

We will now present the steps of an SHR computation.

Definition 8 (*SHR transition*)
Let *Act* be a ranked set of actions. The rank of $a \in Act$ is $\mathrm{ar}(a)$.
A SHR transition is of the form:

$$\Gamma \vdash G \xrightarrow{\Lambda,\pi} \Phi \vdash G'$$

where $\Gamma \vdash G$ and $\Phi \vdash G'$ are well-formed judgements for graphs, $\Lambda : \Gamma \to (Act \times \mathcal{N}^*)$ is a total function and $\pi : \Gamma \to \Gamma$ is an idempotent substitution. Function Λ assigns to each node x the action a and the vector $\vec{y}$ of node references exposed on x by the transition. If $\Lambda(x) = (a, \vec{y})$ then we define $\mathrm{act}_\Lambda(x) = a$ and $\mathrm{n}_\Lambda(x) = \vec{y}$. We require that $\mathrm{ar}(\mathrm{act}_\Lambda(x)) = |\,\mathrm{n}_\Lambda(x)|$, namely the rank of the action must equal the length of the vector.
We define:

- $\mathrm{n}(\Lambda) = \{z | \exists x.z \in \mathrm{n}_\Lambda(x)\}$
  set of exposed names;
- $\Gamma_\Lambda = \mathrm{n}(\Lambda) \setminus \Gamma$
  set of fresh names that are exposed;
- $\mathrm{n}(\pi) = \{x | \exists x' \neq x.\pi(x) = \pi(x')\}$
  set of fused names.

Substitution π allows to merge nodes. Since π is idempotent, it maps every node into a standard representative of its equivalence class. We require that $\forall x \in \mathrm{n}(\Lambda).\pi(x) = x$, i. e. only references to representatives can be exposed. Furthermore we require $\phi \supseteq \pi(\Gamma) \cup \Gamma_\Lambda$, namely nodes are never erased. Nodes in $\Gamma_{Int} = \phi \setminus (\pi(\Gamma) \cup \Gamma_\Lambda)$ are fresh internal nodes, silently created in the transition. We require that no isolated, internal nodes are created, namely $\Gamma_{Int} \subseteq \mathrm{n}(G')$.

Note that the set of names Φ of the resulting graph is fully determined by Γ, Λ, π and G'. Notice also that we can write a SHR transition as

$$\Gamma \vdash G \xrightarrow{\Lambda,\pi} \pi(\Gamma), \Gamma_\Lambda, \Gamma_{Int} \vdash G'.$$

We usually assume to have an action $\epsilon \in \mathit{Act}$ of arity 0 to denote "no synchronization". We may not write explicitly π if it is the identity, and some actions if they are $(\epsilon, \langle\rangle)$. Furthermore we use Λ_ϵ to denote the Λ function that assigns $(\epsilon, \langle\rangle)$ to each node in Γ (note that the dependence on Γ is implicit).

We derive SHR transitions from basic productions using a set of inference rules. Productions define the behaviours of single edges.

Definition 9 (*Production*)
A production is an SHR transition of the form:

$$x_1, \ldots, x_n \vdash L(x_1, \ldots, x_n) \xrightarrow{\Lambda,\pi} \Phi \vdash G$$

where all x_i, $i = 1 \ldots n$ are distinct.
Productions are to be considered as schemas and so they are α-convertible with respect to names in $\{x_1, \ldots, x_n\} \cup \Phi$.

We will now present the set of inference rules for Hoare synchronization. The intuitive idea of Hoare synchronization is that all the edges connected to a node must expose the same action on that node.

Definition 10 (*Rules for Hoare synchronization*)

$$\text{(par)} \quad \frac{\Gamma \vdash G_1 \xrightarrow{\Lambda,\pi} \Phi \vdash G_2 \qquad \Gamma' \vdash G'_1 \xrightarrow{\Lambda',\pi'} \Phi' \vdash G'_2}{\Gamma, \Gamma' \vdash G_1 | G'_1 \xrightarrow{\Lambda \cup \Lambda', \pi \cup \pi'} \Phi, \Phi' \vdash G_2 | G'_2}$$

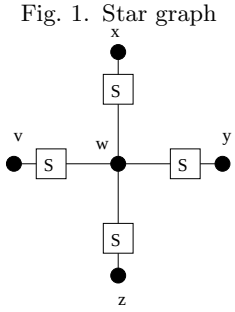
where $(\Gamma \cup \Phi) \cap (\Gamma' \cup \Phi') = \emptyset$.

$$\text{(merge)} \quad \frac{\Gamma \vdash G_1 \xrightarrow{\Lambda,\pi} \Phi \vdash G_2}{\sigma(\Gamma) \vdash \sigma(G_1) \xrightarrow{\Lambda',\pi'} \Phi' \vdash \rho(\sigma(G_2))}$$

where $\sigma : \Gamma \to \Gamma$ is an idempotent substitution and:

(i). $\sigma(x) = \sigma(y) \wedge x \neq y \Rightarrow \Lambda(x) = (a, \vec{v}) \wedge \Lambda(y) = (a, \vec{w})$
(ii). $\rho = \mathrm{mgu}(\{\sigma(\vec{v}) = \sigma(\vec{w}) | \sigma(x) = \sigma(y) \wedge \Lambda(x) = (a, \vec{v}) \wedge \Lambda(y) = (a, \vec{w})\} \cup \{\sigma(x) = \sigma(y) | \pi(x) = \pi(y)\})$
(iii). $\Lambda'(\sigma(z)) = \rho(\sigma(\Lambda(z)))$
(iv). $\pi' = \rho|_{\sigma(\Gamma)}$

Fig. 1. Star graph



$$\text{(idle)} \quad \Gamma \vdash G \xrightarrow{\Lambda_\epsilon, id} \Gamma \vdash G$$

$$\text{(new)} \quad \frac{\Gamma \vdash G_1 \xrightarrow{\Lambda,\pi} \Phi \vdash G_2}{\Gamma, x \vdash G_1 \xrightarrow{\Lambda \uplus \{(x,a,\vec{y})\},\pi} \Phi' \vdash G_2}$$

A transition is obtained by composing productions, which are first applied on disconnected edges, and then by connecting the edges by merging nodes. In particular rule (par) deals with the composition of transitions which use disjoint sets of variables and rule (merge) allows to merge nodes (note that σ is a projection into representatives of equivalence classes). When two nodes are merged, (i) we must have on them the same action, with possibly different tuples of names. Definition (ii) introduces the most general unifier ρ of the union of two sets of equations: the first set identifies (the representatives of) the tuples associated to nodes merged by σ, while the second set of equations is just the kernel of π. Thus ρ is the merge resulting from both π and σ. Note that (iii) Λ is updated with these merges and that (iv) π' is ρ restricted to the nodes of the graph which is the source of the transition. Rule (idle) guarantees that each edge can always make an explicit idle step. Rule (new) allows adding to the source graph an isolated node where arbitrary actions are exposed.

We write $\mathcal{P} \Vdash (\Gamma \vdash G \xrightarrow{\Lambda,\pi} \Phi \vdash G')$ if $\Gamma \vdash G \xrightarrow{\Lambda,\pi} \Phi \vdash G'$ can be obtained from the productions in $\mathcal{P}$ using Hoare inference rules.

We will now present an example of HSHR transition.

Example 1 (*Hirsch et al. 2000*)
We will now show how to use HSHR to derive a 4 elements ring starting from a one element ring, and how we can then specify a reconfiguration that transforms the ring into the star graph in Figure 1.

We use the following productions:

$$x, y \vdash C(x,y) \xrightarrow{(x,\epsilon,\langle\rangle),(y,\epsilon,\langle\rangle)} x, y, z \vdash C(x,z) | C(z,y)$$

$$x, y \vdash C(x,y) \xrightarrow{(x,r,\langle w\rangle),(y,r,\langle w\rangle)} x, y, z \vdash S(y,w)$$

Fig. 2. Productions

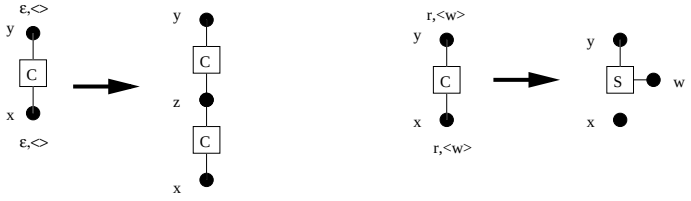
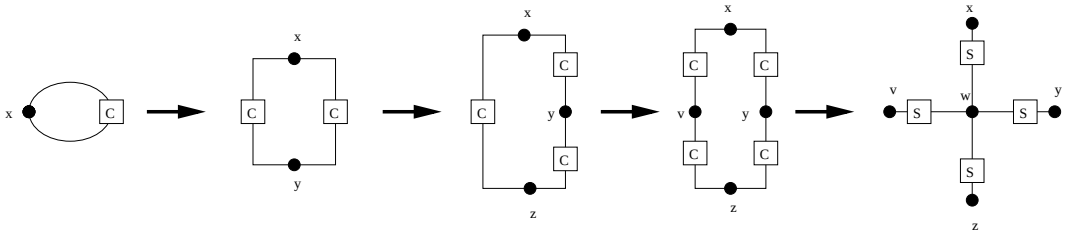


Fig. 3. Ring creation and reconfiguration to star



that are graphically represented in Figure 2. Notice that Λ is represented by decorating every node x in the left hand with $\mathrm{act}_\Lambda(x)$ and $\mathrm{n}_\Lambda(x)$. The first rule allows to create rings, in fact we can create all rings with computations like:

$$x \vdash C(x,x) \to x, y \vdash C(x,y)|C(y,x) \to \\ \to x, y, z \vdash C(x,y)|C(y,z)|C(z,x) \to \\ \to x, y, z, v \vdash C(x,y)|C(y,z)|C(z,v)|C(v,x)$$

In order to perform the reconfiguration into a star we need rules with nontrivial actions, like the second one. This allows to do:

$$x, y, z, v \vdash C(x,y)|C(y,z)|C(z,v)|C(v,x) \to \\ \to x, y, z, v, w \vdash S(x,w)|S(y,w)|S(z,w)|S(v,w)$$

Note that if an edge C is rewritten into an edge S, then all the edges in the ring must use the same production, since they must synchronize via action r. They must agree also on $\mathrm{n}_\Lambda(x)$ for every x, thus all the newly created nodes are merged. The whole transition is represented in Figure 3.

It is easy to show that if we can derive a transition T, then we can also derive every transition obtainable from T by applying an injective renaming.

Lemma 1
Let $\mathcal{P}$ be a set of productions and σ an injective substitution.
$\mathcal{P} \Vdash (\Gamma \vdash G \xrightarrow{\Lambda,\pi} \Phi \vdash G')$ iff:
$\mathcal{P} \Vdash (\sigma(\Gamma) \vdash \sigma(G) \xrightarrow{\Lambda',\pi'} \sigma(\Phi) \vdash \sigma(G'))$

where $\Lambda'(\sigma(x)) = \sigma(\Lambda(x))$ and $\pi'(\sigma(x)) = \sigma(\pi(x))$.

Proof
By rule induction.

2.3 Logic programming

In this paper we are not interseted in logic computations as refutations of goals for problem solving or artificial intelligence, but we consider logic programming as a *goal rewriting mechanism*. We can consider logic subgoals as concurrent communicating processes that evolve according to the rules defined by the clauses and that use *unification* as the fundamental interaction primitive. A presentation of this kind of use of logic programming can be found in Bruni et al. (2001).

In order to stress the similarities between logic programming and process calculi we present a semantics of logic programming based on a labelled transition system.

Definition 11
We have for clauses (C) and goals (G) the following grammar:

$$C ::= A \leftarrow G$$

$$G ::= G, G \mid A \mid \Box$$

where A is a logical atom, "," is the AND conjunction and $\Box$ is the empty goal. We can assume "," to be associative and commutative and with unit $\Box$.

The informal semantics of $A \leftarrow B_1, \ldots, B_n$ is "for every assignment of the variables, if $B_1, \ldots, B_n$ are all true, then A is true".

A logic program is a set of clauses.

Definition 12 (*Partial SLD-derivation*)
Let P be a logic program.
We define a step of an SLD-resolution computation using the following rules:

$$\frac{H \leftarrow B_1, \ldots, B_k \in P \qquad \theta = \mathrm{mgu}(A, H\rho)}{P \Vdash A \xrightarrow{\theta} B_1, \ldots, B_k\rho\theta} \qquad \text{atomic goal}$$

where ρ is an injective renaming of variables such that all the variables in the clause variant $\rho(H \leftarrow B_1, \ldots, B_k)$ are fresh.

$$\frac{P \Vdash G \xrightarrow{\theta} F}{P \Vdash G, G' \xrightarrow{\theta} F, G'\theta} \qquad \text{conjunctive goal}$$

We will omit $P \Vdash$ if P is clear from the context.
A partial SLD-derivation of $P \cup \{G\}$ is a sequence (eventually empty) of steps of SLD-resolution allowed by program P with initial goal G.

3 Mapping Fusion Calculus into Synchronized Hyperedge Replacement

In this section we present a mapping from Fusion Calculus to HSHR.
This mapping is quite complex since there are many differences between the two formalisms. First of all we need to bridge the gap between a process calculus and a graph transformation formalism, and this is done by associating edges to sequential processes and by connecting them according to the structure of the system. Moreover we need to map Milner synchronization, which is used in Fusion Calculus, into Hoare synchronization. In order to do this we define some connection structures that we will call amoeboids which implement Milner synchronization using Hoare connectors. An amoeboid is essentially a router that connects an action with the corresponding coaction. This is possible since in HSHR an edge can do many synchronizations on different nodes at the same time. Finally some restrictions have to be imposed on HSHR in order to have an interleaving behaviour as required by Fusion Calculus.

We define the translation on processes in the form $(\vec{x})P$ where P is the parallel composition of sequential processes. Notice that every process can be reduced to the above form by applying the structural axioms: recursive definitions which are not inside a sequential agent have to be unfolded once and scope operators which are not inside a sequential agent must be taken to the outside.

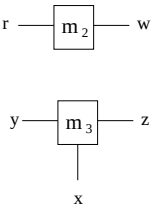
We define the translation also in the case $(\vec{x})P$ is not closed w. r. t. names (but it must be closed w. r. t. process variables) since this case is needed for defining productions.

In the form $(\vec{x})P$ we assume that the ordering of names in $(\vec{x})$ is fixed, dictated by some structural condition on their occurrences in P.
For our purposes, it is also convenient to express process P in $(\vec{x})P$ as $P = P'\sigma$, where P' is a linear agent, i. e. every name in it appears once. We assume that the free names of P' are fresh, namely $\mathrm{fn}(P') \cap \mathrm{fn}(P) = \emptyset$, and again structurally ordered. The corresponding vector is called $\mathrm{fnarray}(P')$.

The decomposition $P = P'\sigma$ highlights the role of amoeboids. In fact, in the translation, substitution σ is made concrete by a graph consisting of amoeboids, which implement a router for every name in $\mathrm{fn}(P)$. More precisely, we assume the existence of edge labels m_i and n of ranks $i = 2, 3, \ldots$ and 1 respectively. Edges labelled by m_i implement routers among i nodes, while n edges "close" restricted names $\vec{x}$ in $(\vec{x})P'\sigma$.

Finally, linear sequential processes S in P' must also be given a standard form. In fact they will be modeled in the HSHR translation by edges labelled by L_S, namely by a label encapsulating S itself. However in the derivatives of a recursive process the same sequential process can appear with different variables an unbound number of times. To make the number of labels (and also of productions, as we will

Fig. 4. Amoeboids for σ



see in short) finite, for every given process, we choose standard names $x_1, \ldots, x_n$ and order them structurally:
$S = \hat{S}(x_1, \ldots, x_n)\rho_S$ with $S_1 = S_2\rho$ implying $\hat{S}_1 = \hat{S}_2$ and $\rho_{S_1} = \rho \cdot \rho_{S_2}$.

We can now define the translation from Fusion Calculus to HSHR. The translation is parametrized by the nodes in the vectors $\vec{v}$ and $\vec{w}$ we choose to represent the names in $\vec{x}$ and $\mathrm{fnarray}(P')$.

Definition 13 (*Translation from Fusion Calculus to HSHR*)
$[\![(\vec{x})P\sigma]\!]_{\vec{v},\vec{w}} = \Gamma \vdash |_{x\in\vec{v}} n(x) | [\![P]\!]\{\vec{w}/\,\mathrm{fnarray}(P)\} | [\![\sigma]\!]\{\vec{v}/\vec{x}\}\{\vec{w}/\,\mathrm{fnarray}(P)\}$
where:
$|\vec{v}| = |\vec{x}|$,
$|\vec{w}| = |\,\mathrm{fnarray}(P)|$,
$\vec{v} \cap \vec{w} = \emptyset$
and with:
$\Gamma = \mathrm{fn}((\vec{x})P\sigma), \vec{v}, \vec{w}$.

$[\![0]\!] = nil$
$[\![S]\!] = L_{\hat{S}}(x_1, \ldots, x_n)\rho_S$
$[\![P_1|P_2]\!] = [\![P_1]\!] | [\![P_2]\!]$

$[\![\sigma]\!] = |_{x\in \mathrm{Im}(\sigma)} m_{k+1}(x, \sigma^{-1}(x))$ where $k = |\sigma^{-1}(x)|$.

In the above translation, graph $[\![P]\!]$ consists of a set of disconnected edges, one for each sequential process of $(\vec{x})P\sigma$. Also the translation produces a graph with three kinds of nodes. The nodes of the first kind are those in $\vec{w}$. Each of them is adjacent to exactly two edges, one representing a sequential process of P, and the other an amoeboid. Also the nodes in $\vec{v}$ are adjacent to two edges, an amoeboid and an n edge. Finally the nodes in $\mathrm{fn}((\vec{x})P\sigma)$ are adjacent only to an amoeboid.

As mentioned above, translation $[\![\sigma]\!]$ builds an amoeboid for every free name x of $P\sigma$: it has $k+1$ tentacles, where k are the occurrences of x in $P\sigma$, namely the free names of P mapped to it. Notice that the chocie of the order within $\sigma^{-1}(x)$ is immaterial, since we will see that amoeboids are commutative with respect to their tentacles. However, to make the translation deterministic, $\sigma^{-1}(x)$ could be ordered according to some fixed precedence of the names.

Example 2 (*Translation of a substitution*)
Let $\sigma = \{y/x, y/z, r/w\}$. The translation of σ is in figure 4.

Example 3 (*Translation of a process*)
Let us consider the (closed) process $(uz)\overline{u}z.0|\operatorname{rec} X.(x)ux.(\overline{u}x.0|X)$. We can write it in the form $(\vec{x})P$ as:
$(uzy)\overline{u}z.0|uy.(\overline{u}y.0|\operatorname{rec} X.(x)ux.(\overline{u}x.0|X))$
Furthermore we can decompose P into $P'\sigma$ where:
$P' = \overline{y_1}y_2.0|y_3y_4.(\overline{y_5}y_6.0|\operatorname{rec} X.(x)y_7x.(\overline{y_8}x.0|X))$
$\sigma = \{u/y_1, z/y_2, u/y_3, y/y_4, u/y_5, y/y_6, u/y_7, u/y_8\}$.
We can now perform the translation which is parametrized with $\vec{v} = (u, z, y)$ and $\vec{w} = (y_1, y_2, y_3, y_4, y_5, y_6, y_7, y_8)$:

$$\begin{aligned}&[\![(\vec{x})P'\sigma]\!]_{\vec{v},\vec{w}} = u, z, y, y_1, y_2, y_3, y_4, y_5, y_6, y_7, y_8 \vdash \\ &\quad L_{\overline{x_1}x_2.0}(y_1, y_2)|L_{x_1x_2.(\overline{x_3}x_4.0|\operatorname{rec} X.(x)x_5x.(\overline{x_6}x.0|X))}(y_3, y_4, y_5, y_6, y_7, y_8)| \\ &\qquad m_6(u, y_1, y_3, y_5, y_7, y_8)|m_2(z, y_2)|m_3(y, y_4, y_6)|n(u)|n(z)|n(y)\end{aligned}$$

Now we define productions used in the HSHR system.
We have two kinds of productions: auxiliary productions that are applied to amoeboid edges and process productions that are applied to process edges.
Before showing process productions we need to present the translation from Fusion Calculus prefixes into HSHR transition labels.

Definition 14
The translation from Fusion Calculus prefixes into HSHR transtion labels is the following:
$[\![\alpha]\!] = (\Lambda, \pi)$ where
if $\alpha = u\vec{x}$ then $\Lambda(u) = (in_n, \vec{x})$, $\Lambda(x) = (\epsilon, \langle\rangle)$ if $x \neq u$ with $n = |\vec{x}|$, $\pi = id$
if $\alpha = \overline{u}\vec{x}$ then $\Lambda(u) = (out_n, \vec{x})$, $\Lambda(x) = (\epsilon, \langle\rangle)$ if $x \neq u$ and $n = |\vec{x}|$, $\pi = id$
if $x = \phi$ then $\Lambda = \Lambda_\epsilon$ and π is any substitutive effect of ϕ.
We will write $[\![u\vec{x}]\!]$ and $[\![\overline{u}\vec{x}]\!]$ as $(u, in_n, \vec{x})$ and $(u, out_n, \vec{x})$ respectively.

Definition 15 (*Process productions*)
We have a process production for each prefix at the top level of a linear standard sequential process.
Let $\sum_i \alpha_i.P_i$ be such a process. Its productions can be derived with the following inference rule:

$$\frac{[\![P_i]\!]_{\vec{v},\vec{w}} = \Gamma \vdash G}{x_1, \ldots, x_n \vdash L_{\sum_i \alpha_i.P_i}(x_1, \ldots, x_n) \xrightarrow{[\![\alpha_i]\!]} \Gamma, \Gamma' \vdash |_{x \in \Gamma''} n(x)|G}$$

if $(\vec{v} \cup \vec{w}) \cap \{x_1, \ldots, x_n\} = \emptyset$
with $\Gamma' = \pi(x_1, \ldots, x_n) \setminus \Gamma$ and $\Gamma'' = \Gamma' \setminus (\mathrm{n}(\Lambda) \cup \mathrm{n}(\pi))$.

We add some explanations on the derivable productions. Essentially, if $\alpha_i.P_i$ is a possible choice, the edge labelled by the process can have a transition labelled by α_i to something related to $[\![P_i]\!]_{\vec{v},\vec{w}}$. If α_i is a fusion ϕ, according to the semantics of the calculus, a substitutive effect π of it should be applied to P_i before translating it. However, being $\phi.P_i$ linear, the names occurring in ϕ do not occur in P_i, i. e. $P_i\pi = P_i$. Furthermore, $\Gamma \vdash G$ must be enriched in two ways: since nodes can never

be erased, nodes which are present in the sequential process and then forgotten, i.e. the nodes in Γ', must be added to Γ. Also "close" edges must be associated to forgotten nodes, provided they are not exposed, i.e. to nodes in Γ''. Finally, notice that the nodes in $\vec{w}$ and $\vec{v}$, which implement routers and represent names restricted in P_i, respectively, must be fresh, namely different from all the other existing names.

Note that when translating the RHS $(\vec{x})P\sigma$ of productions we may have variables in $P\sigma$ which occur just once: they are also renamed by σ. In this case we will have in the translation some m_2 connectors which are not essential. In the examples we will avoid these connectors (the nodes to be connected are instead merged).

Example 4 (*Translation of a production*)
Let us consider firstly the simple agent $\overline{x_1}x_2.0$.
The only production for this agent is:
$x_1, x_2 \vdash L_{\overline{x_1}x_2.0}(x_1, x_2) \xrightarrow{(x_1, out_1, x_2)} x_1, x_2 \vdash n(x_1)$
where we closed node x_1 but not node x_2 since the second one is exposed on x_1.
Let us consider a more complex example:
$x_1x_2.(\overline{x_3}x_4.0|recX.(x)x_5x.(\overline{x_6}x.0|X))$.
The process $\overline{x_3}x_4.0|recX.(x)x_5x.(\overline{x_6}x.0|X)$ can be transformed into:
$(y)\overline{y_1}y_2.0|y_3y_4.(\overline{y_5}y_6.0|recX.((x)y_7x.(\overline{y_8}x.0|X)))\sigma$
where $\sigma = \{x_3/y_1, x_4/y_2, x_5/y_3, y/y_4, x_6/y_5, y/y_6, x_5/y_7, x_6/y_8\}$. Its translation is:

$$x_3, x_4, x_5, x_6, y_3, y_4, y_5, y_6, y_7, y_8, y \vdash$$
$$L_{\overline{x_1}x_2.0}(x_3, x_4)|L_{x_1x_2.(\overline{x_3}x_4.0|recX.((x)x_5x.(\overline{x_6}x.0|X)))}(y_3, y_4, y_5, y_6, y_7, y_8)|$$
$$m_3(x_5, y_3, y_7)|m_3(x_6, y_5, y_8)|m_3(y, y_4, y_6)|n(y)$$

Thus the production is:

$$x_1, x_2, x_3, x_4, x_5, x_6 \vdash L_{x_1x_2.(\overline{x_3}x_4.0|recX.(x)x_5x.(\overline{x_6}x.0|X))}(x_1, x_2, x_3, x_4, x_5, x_6)$$
$$\xrightarrow{x_1, in_1, x_2}$$
$$x_1, x_2, x_3, x_4, x_5, x_6, y_3, y_4, y_5, y_6, y_7, y_8, y \vdash$$
$$L_{\overline{x_1}x_2.0}(x_3, x_4)|L_{x_1x_2.(\overline{x_3}x_4.0|recX.((x)x_5x.(\overline{x_6}x.0|X)))}(y_3, y_4, y_5, y_6, y_7, y_8)|$$
$$m_3(x_5, y_3, y_7)|m_3(x_6, y_5, y_8)|m_3(y, y_4, y_6)|n(y)|n(x_1)$$

We will now show the productions for amoeboids.

Definition 16 (*Auxiliary productions*)
We have an auxiliary production of the form:

$$\Gamma \vdash m_k(\dots, x_1, \dots, x_2, \dots) \xrightarrow{(x_1, in_n, \vec{y}_1)(x_2, out_n, \vec{y}_2)}$$
$$\Gamma, \vec{y}_1, \vec{y}_2 \vdash |_{i=1\dots|\vec{y}_1|} m_2(y_{1,i}, y_{2,i})|m_k(\dots, x_1, \dots, x_2, \dots)$$

for each k and n and each pair of nodes x_1 and x_2 in Γ. Here $\vec{y}_1$ and $\vec{y}_2$ are two vectors of fresh names such that $|\vec{y}_1| = |\vec{y}_2| = n$.
Note that we also have the analogous production where x_1 and x_2 are swapped. In

particular the set of productions for a m_k edge is invariant with respect to permutations of the tentacles, modeling the fact that tentacles are essentially unordered. We have no productions for edges labelled with n.

The notion of amoeboid introduced previously is not sufficient for our purposes. In fact existing amoeboids can be connected using m_2 edges and nodes that are no more used can be closed using n edges. Thus we will present a more general definition of amoeboid for a set of nodes and we will show that in the situations of interest these amoeboids behave exactly as the simpler m_i edges.

Definition 17 (*Structured amoeboid*)
Given a set of nodes S, a structured amoeboid $M(x_1, \ldots, x_n)$ for S is any connected graph composed by m and n edges that satisfies the following properties:

- the nodes are either internal or external;
- S is the set of external nodes;
- external nodes are connected to exactly one edge of the amoeboid;
- internal nodes are connected to exactly two edges of the amoeboid.

We consider *equivalent* all the amoeboids with the same set S of external nodes. Note that $m_{|S|}(\vec{s})$ where the set of names in $\vec{s}$ is S is an amoeboid for S.

Lemma 2
If $M(x_1, \ldots, x_n)$ is a structured amoeboid, the only possible transitions for $M(x_1, \ldots, x_n)$ which are non idle and expose non ϵ actions on at most two nodes $x_j, x_k \in \{x_1, \ldots, x_n\}$ do not depend on the structure of M and are of the form:

$$x_1, \ldots, x_n \vdash M(x_1, \ldots, x_n) \xrightarrow{\Lambda, id} x_1, \ldots, x_n, \vec{y}_1, \vec{y}_2 \vdash |_{i=1\ldots|\vec{y}_1|} m_2(y_{1,i}, y_{2,i}) | M(x_1, \ldots, x_n)$$

where $\Lambda(x_j) = (in_n, \vec{y}_1)$ and $\Lambda(x_k) = (out_n, \vec{y}_2)$.

Proof
By induction on the number of edges in the amoeboid.

Thanks to the above result we will refer to structured amoeboids simply as amoeboids.

We can now present the results on the correctness and completeness of our translation.

Theorem 2 (*Correctness*)
For each closed fusion process P, if $P \to P'$ then there exist Λ, Γ and G such that $[\![P]\!]_{\vec{v},\vec{w}} \xrightarrow{\Lambda, id} \Gamma \vdash G$. Furthermore $\Gamma \vdash G$ is equal to $[\![P']\!]_{\vec{v'},\vec{w'}}$ up-to isolated nodes, up to injective renamings and up-to equivalence of amoeboids ($\Gamma \vdash G$ can have a structured amoeboid where $[\![P']\!]_{\vec{v'},\vec{w'}}$ has a simple one).

Proof
The proof is by rule induction on the reduction semantics.
See Appendix A.

Theorem 3 (*Completeness*)

For each closed fusion process P if $[\![P]\!]_{\vec{v},\vec{w}} \xrightarrow{\Lambda,\pi} \Gamma \vdash G$ with an HSHR transition that uses exactly two productions for communication or one production for a fusion action (plus any number of auxiliary productions) then $P \rightarrow P'$ and $\Gamma \vdash G$ is equal to $[\![P']\!]_{\vec{v'},\vec{w'}}$ up-to isolated nodes, up-to injective renamings and up-to equivalence of amoeboids ($\Gamma \vdash G$ can have a structured amoeboid where $[\![P']\!]_{\vec{v'},\vec{w'}}$ has a simple one).

Proof

See Appendix A.

These two theorems prove that the allowed transitions in the HSHR setting correspond to reductions in the Fusion Calculus setting. Note that in HSHR we must consider only transitions where we have either two productions for communication or one production for a fusion action. This is necessary to model the interleaving behaviour of Fusion Calculus within the HSHR formalism, which is concurrent. Note that the differences between the final graph of a transition and the translation of the final process of a Fusion Calculus reduction are not important, since the two graphs have essentially the same behaviours (see lemma 1 for the effect of an injective renaming and lemma 2 for the characterization of the behaviour of a complex amoeboid).

Note that in the HSHR model the behavioural part of the system is represented by productions while the topological part is represented by graph. Thus we have a convenient separation between the two different aspects.

Example 5 (*Translation of a transition*)

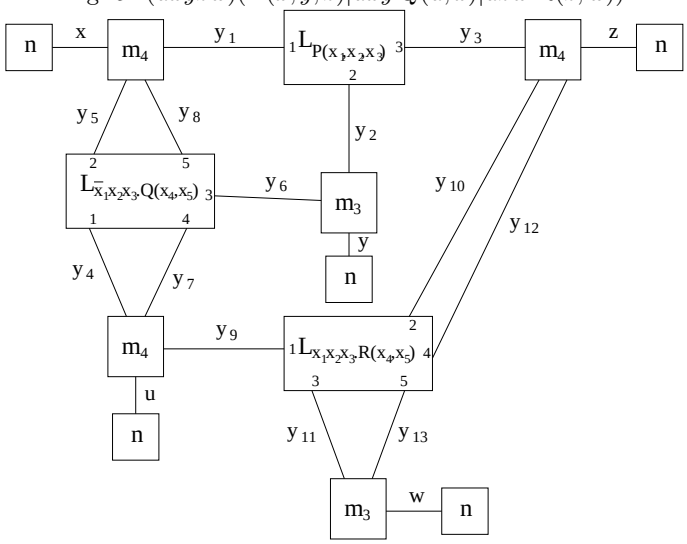
We will now show an example of the translation.
Let us consider the process:
$(uxyzw)(P(x,y,z)|\overline{u}xy.Q(u,x)|uzw.R(z,w))$.
Note that it is already in the form $(\vec{x})P$.
It can do the following transition:

$$(uxyzw)(P(x,y,z)|\overline{u}xy.Q(u,x)|uzw.R(z,w)) \rightarrow (uxy)(P(x,y,z)|Q(u,x)|R(z,w))\{x/z, y/w\}$$

We can write P in the form:
$(P(y_1,y_2,y_3)|\overline{y_4}y_5y_6.Q(y_7,y_8)|y_9y_{10}y_{11}.R(y_{12},y_{13}))\sigma$

where:
$\sigma = \{x/y_1, y/y_2, z/y_3, u/y_4, x/y_5, y/y_6, u/y_7, x/y_8, u/y_9, z/y_{10}, w/y_{11}, z/y_{12}, w/y_{13}\}$.

Fig. 5. $(uxyzw)(P(x,y,z)|\overline{u}xy.Q(u,x)|uzw.R(z,w))$



A translation of the starting process is:

$$u,x,y,w,z,y_1,y_2,y_3,y_4,y_5,y_6,y_7,y_8,y_9,y_{10},y_{11},y_{12},y_{13} \vdash$$
$$L_{P(x_1,x_2,x_3)}(y_1,y_2,y_3)|L_{\overline{x_1}x_2x_3.Q(x_4,x_5)}(y_4,y_5,y_6,y_7,y_8)|$$
$$L_{x_1x_2x_3.R(x_4,x_5)}(y_9,y_{10},y_{11},y_{12},y_{13})|m_4(u,y_4,y_7,y_9)|m_4(x,y_1,y_5,y_8)|$$
$$m_3(y,y_2,y_6)|m_4(z,y_3,y_{10},y_{12})|m_3(w,y_{11},y_{13})|n(u)|n(x)|n(y)|n(w)|n(z)$$

A graphical representation is in figure 5.
We have the following process productions:

$$y_1,y_2,y_3,y_4,y_5 \vdash L_{\overline{x_1}x_2x_3.Q(x_4,x_5)}(y_1,y_2,y_3,y_4,y_5) \xrightarrow{(y_1,out_2,\langle y_2,y_3\rangle)}$$
$$y_1,y_2,y_3,y_4,y_5 \vdash L_{Q(x_1,x_2)}(y_4,y_5)|n(y_1)$$

$$y_1,y_2,y_3,y_4,y_5 \vdash L_{x_1x_2x_3.R(x_4,x_5)}(y_1,y_2,y_3,y_4,y_5) \xrightarrow{(y_1,in_2,\langle y_2,y_3\rangle)}$$
$$y_1,y_2,y_3,y_4,y_5 \vdash L_{R(x_1,x_2)}(y_4,y_5)|n(y_1)$$

In order to apply (suitable variants of) these two productions concurrently we have to synchronize their actions. This can be done since in the actual transition actions are exposed on nodes y_4 and y_9 respectively, which are connected to the same m_4 edge. Thus the synchronization can be performed (see figure 6) and we obtain as final graph:

$$u,x,y,w,z,y_1,y_2,y_3,y_4,y_5,y_6,y_7,y_8,y_9,y_{10},y_{11},y_{12},y_{13} \vdash$$
$$L_{P(x_1,x_2,x_3)}(y_1,y_2,y_3)|L_{Q(x_1,x_2)}(y_7,y_8)|n(y_4)|L_{R(x_1,x_2)}(y_{12},y_{13})|n(y_9)|$$
$$m_4(u,y_4,y_7,y_9)|m_4(x,y_1,y_5,y_8)|m_3(y,y_2,y_6)|m_4(z,y_3,y_{10},y_{12})|m_3(w,y_{11},y_{13})|$$
$$m_2(y_5,y_{10})|m_2(y_6,y_{11})|n(u)|n(x)|n(y)|n(w)|n(z)$$

Fig. 6. Graph with actions

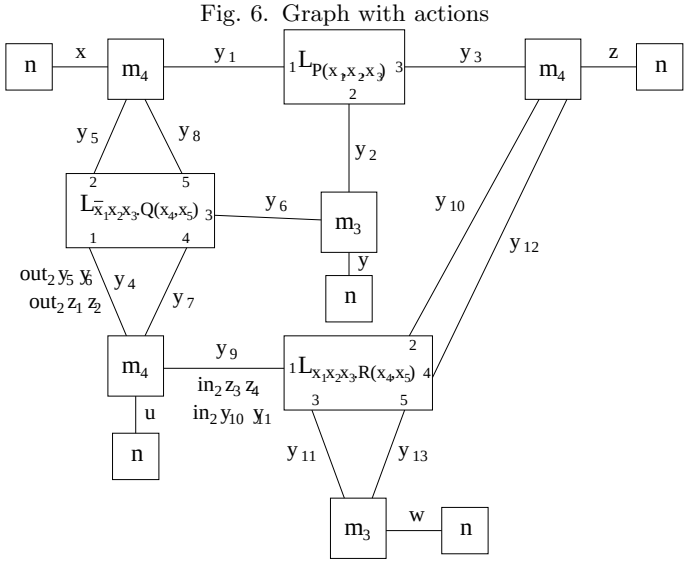
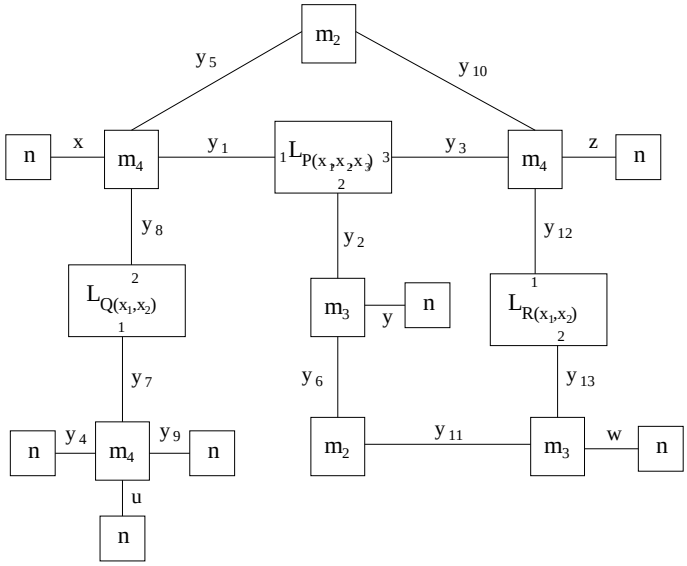


Fig. 7. Resulting graph



which is represented in figure 7.

The amoeboids connect the following tuples of nodes:
(u, y_4, y_7, y_9), $(x, y_1, y_5, y_8, y_{10}, z, y_3, y_{12})$, $(w, y_{11}, y_{13}, y_6, y, y_2)$ thus if we connect these sets of nodes with simple amoeboids instead of with complex ones we have up-to injective renamings a translation of $(uxy)P(x, y, x)|Q(u, x)|R(x, y)$ as required.

Example 6 (*Translation of a transition with recursion*)
We will show here an example that uses recursion. Let us consider the closed process $(uz)\overline{u}z|recX.(x)ux.(\overline{u}x.0|X)$. The translation of this process, as shown in example

3 is:

$$u, z, y, y_1, y_2, y_3, y_4, y_5, y_6, y_7, y_8 \vdash$$
$$L_{\overline{x_1}x_2.0}(y_1, y_2) | L_{x_1x_2.(\overline{x_3}x_4.0|\,\mathrm{rec}\,X.(x)x_5x.(\overline{x_6}x.0|X))}(y_3, y_4, y_5, y_6, y_7, y_8) |$$
$$m_6(u, y_1, y_3, y_5, y_7, y_8) | m_2(z, y_2) | m_3(y, y_4, y_6) | n(u) | n(z) | n(y)$$

We need the productions for two sequential edges (for the first step): $\overline{x_1}x_2.0$ and $x_1x_2.(\overline{x_3}x_4.0|recX.(x)x_5x.(\overline{u}x_6.0|X))$.
The productions are the ones of example 4 (we write them here in a suitable α-converted form):

$$y_1, y_2 \vdash L_{\overline{x_1}x_2.0}(y_1, y_2) \xrightarrow{(y_1, out_1, y_2)} y_1, y_2 \vdash n(y_1)$$

$$y_3, y_4, y_5, y_6, y_7, y_8 \vdash L_{x_1x_2.(\overline{x_3}x_4.0|recX.(x)x_5x.(\overline{x_6}x.0|X))}(y_3, y_4, y_5, y_6, y_7, y_8)$$
$$\xrightarrow{y_3, in_1, y_4}$$
$$y_3, y_4, y_5, y_6, y_7, y_8, z_1, z_2, z_3, z_4, z_5, z_6, y' \vdash$$
$$L_{\overline{x_1}x_2.0}(y_5, y_6) | L_{x_1x_2.(\overline{x_3}x_4.0|recX.((x)x_5x.(\overline{x_6}x.0|X)))}(z_1, z_2, z_3, z_4, z_5, z_6) |$$
$$m_3(y_7, z_1, z_5) | m_3(y_8, z_3, z_6) | m_3(y', z_2, z_4) | n(y') | n(y_3)$$

By using (variants of) these two productions and a production for m_6 (the other edges stay idle) we have the following transition:

$$u, z, y, y_1, y_2, y_3, y_4, y_5, y_6, y_7, y_8 \vdash$$
$$L_{\overline{x_1}x_2.0}(y_1, y_2) | L_{x_1x_2.(\overline{x_3}x_4.0|\,\mathrm{rec}\,X.(x)x_5x.(\overline{x_6}x.0|X))}(y_3, y_4, y_5, y_6, y_7, y_8) |$$
$$m_6(u, y_1, y_3, y_5, y_7, y_8) | m_2(z, y_2) | m_3(y, y_4, y_6) | n(u) | n(z) | n(y)$$
$$\xrightarrow{(y_1, out_1, y_2)(y_3, in_1, y_4)}$$
$$u, y, z, x, y_1, y_2, y_3, y_4, y_5, y_6, y_7, y_8, z_2, z_3, z_4, z_5, z_6, z_7, z_8, y' \vdash n(y_1) | L_{\overline{x_1}x_2.0}(y_5, y_6) |$$
$$L_{x_1x_2.(\overline{x_3}x_4.0|recX.((x)x_5x.(\overline{x_6}x.0|X)))}(z_1, z_2, z_3, z_4, z_5, z_6) |$$
$$m_3(y_7, z_1, z_5) | m_3(y_8, z_3, z_6) | m_3(y', z_2, z_4) | n(y') | n(y_3) |$$
$$m_6(u, y_1, y_3, y_5, y_7, y_8) | m_2(y_2, y_4) | m_2(z, y_2) | m_3(y, y_4, y_6) | n(u) | n(y) | n(z)$$

The resultling graph is up-to injective renaming a suitable translation of:
$(uyy')(\overline{u}y.0|uy'.(\overline{u}y'.0|recX.(x)ux.(\overline{u}x.0|X)))$
as required.

We end this section with a simple schema on the correspondence between the two models.

Fusion	HSHR	Fusion	HSHR
Closed process	Graph	Reduction	Transition
Sequential process	Edge	Name	Amoeboid
Prefix execution	Production	0	Nil

4 Mapping Hoare SHR into logic progamming

We will now present a mapping from HSHR into a subset of logic programming called Synchronized Logic Programming (SLP). The idea is to compose this mapping with the previous one obtaining a mapping from Fusion Calculus into logic programming.

4.1 The mapping

In this subsection we will present Synchronized Logic Programming.
SLP has been introduced because logic programming allows for many execution strategies and for complex interactions and data structures and so it is necessary to limit it in order to keep only the computations which have an HSHR correspondent.

We will impose three kinds of restrictions:

- we use only goals that correspond to graphs (goal-graphs);
- we allow in programs only clauses that correspond to graph synchronizations (synchronized clauses);
- we choose only computations that model HSHR transitions (synchronized computations).

Definition 18 (*Goal-graph*)
We call goal-graph a goal which has no functional symbols.

Definition 19 (*Synchronized program*)
A synchronized program is a finite set of synchronized rules, i. e. definite program clauses such that:

- the body of each rule is a goal-graph;
- the head of each rule is $A(t_1, \ldots, t_n)$ where t_i is either a variable or a single function symbol applied to variables. If it is a variable then it also appears in the body of the clause.

Example 7

$q(f(x), y) \leftarrow p(x, y)$	synchronized rule;
$q(f(x), y) \leftarrow p(x, f(y))$	not synchronized because $p(x, f(y))$ is not a goal-graph;
$q(g(f(x)), y) \leftarrow p(x, y)$	not synchronized because it contains nested functions;
$q(f(x), y, f(z)) \leftarrow p(x)$	not synchronized because y is an argument of the head predicate but it does not appear in the body;
$q(f(x), f(z)) \leftarrow p(x)$	synchronized rule, even if z does not appear in the body.

In our approach, a clause $A(t_1, \ldots, t_n) \leftarrow B_1, \ldots, B_n$ represents a production where the head predicate A is the label of the edge in the left hand side, and the body $B_1, \ldots, B_n$ is the graph in the right hand side. Term t_i in the head represents the action occurring in x_i, if $A(x_1, \ldots, x_n)$ is the edge matched by the

production. Intuitively, the first condition of Definition 19 says that the result of a local rewriting must be a goal-graph. The second condition forbids synchronizations with structured actions, which are not allowed in HSHR. Furthermore it imposes that we cannot disconnect from a node without synchronizing on it [3].

Now we will define the subset of computations we are interested in.

Definition 20 (*Synchronized Logic Programming*)
Given a synchronized program P we write:

$$G_1 \stackrel{\theta}{\Rightarrow} G_2$$

iff $G_1 \stackrel{\theta'}{\longrightarrow}{}^{*} G_2$ and all steps performed in the computation expand different atoms of G_1, $\theta'|_{\mathrm{n}(G_1)} = \theta$ and both G_1 and G_2 are goal-graphs.
We call $G_1 \stackrel{\theta}{\Rightarrow} G_2$ a big-step and all the $\rightarrow$ steps in a big-step small-steps.
A SLP computation is:
$G_1 \Rightarrow^* G_2$ i. e. a sequence of 0 or more big-steps.

4.2 The mapping

We want to use SLP to model HSHR systems. As a first step we need to translate graphs, i. e. syntactic judgements, to goals. In this translation, edge labels will be mapped into SLP predicates. Goals corresponding to graphs will have no function symbols. However function symbols will be used to represent actions. In the translation we will lose the context Γ.

Definition 21 (*Translation for syntactic judgements*)
We define the translation operator $[\![-]\!]$ as:

$$[\![\Gamma \vdash L(x_1, \ldots, x_n)]\!] = L(x_1, \ldots, x_n)$$
$$[\![\Gamma \vdash G_1 | G_2]\!] = [\![\Gamma \vdash G_1]\!], [\![\Gamma \vdash G_2]\!]$$
$$[\![\Gamma \vdash nil]\!] = \Box$$

Sometimes we will omit the Γ part of the syntactic judgement. We can do this because it does not influence the translation.

Definition 22
Let $\Gamma \vdash G$ and $\Gamma' \vdash G'$ be graphs. We define the equivalence relation $\cong$ in the following way: $\Gamma \vdash G \cong \Gamma' \vdash G'$ iff $G \equiv G'$.

[3] This condition has only the technical meaning of making impossible some rewritings in which an incorrect transition may not be forbidden because its only effect is on the discarded variable. Luckily, we can impose this condition without altering the power of the formalism, because we can always perform a special foo action on the node we leave and make sure that all the other edges can freely do the same action. For example we can rewrite $q(f(x), y, f(z)) \leftarrow p(x)$ as $q(f(x), foo(y), f(z)) \leftarrow p(x)$, which is an allowed synchronized rule.

Observe that if two judgements are equivalent then they can be written as:
$\Gamma, \Gamma_{unused} \vdash G$
$\Gamma, \Gamma'_{unused} \vdash G$
where $\Gamma = \mathrm{n}(G)$.

Theorem 4 (*Correspondence of judgements and goal-graphs*)
The operator $[\![-]\!]$ defines an isomorphism between judgements (defined up to $\cong$) and goal-graphs.

Proof
The proof is straightforward observing that syntactic rules for graph judgements impose only conditions on Γ, the operator $[\![-]\!]$ defines a bijection between representatives of syntactic judgements and representatives of goal-graphs and the congruence on the two structures is essentially the same. □

We now define the translation from HSHR productions to definite clauses.

Definition 23 (*Translation from productions to clauses*)
We define the translation operator $[\![-]\!]$ as:

$$[\![L(x_1, \ldots, x_n) \xrightarrow{\Lambda,\pi} G]\!] = L(a_1(\pi(x_1), \vec{y}_1), \ldots, a_n(\pi(x_n), \vec{y}_n)) : - [\![G]\!]$$

if $\Lambda = \{(x_i, a_i, \vec{y}_i)\}_{i=1\ldots n}$ and $a_i \neq \epsilon$. If $a_i = \epsilon$ we write simply $\pi(x_i)$ instead of $\epsilon(\pi(x_i))$.

The idea of the translation is that the condition given by an action $(x, a, \vec{y})$ is represented by using the term $a(\pi(x), \vec{y})$ as argument in the position that corresponds to x. Notice that in this term a is a function symbol, while π is a subsitution which must be applied to its argument. During unification x will be bound to that term and when other instances of x are met, the corresponding term must have the same functional symbol (as required by Hoare synchronization) in order to be unifiable. Furthermore the corresponding tuples of transmitted nodes are unified. Since x will disappear we need another name to represent the node that corresponds to x. We use the first argument of a to this purpose. If two nodes are merged by π then their successors are the same as required.

Observe that we do not need to translate all the possible variants of the rules since variants with fresh names are automatically built when the clauses are applied, but we have to translate also rules corresponding to (idle) transitions of our edges[4]. Notice also that the clauses we obtain are synchronized clauses.

The observable substitution contains information on Λ and π. Thus given a transition we can associate to it a substitution θ. We have different choices for θ according to where we map names on which we expose non ϵ actions. In fact in HSHR nodes are mapped to their representatives according to π while in SLP θ cannot

[4] This is necessary since we want be able to change the variables that correspond to nodes during idle transitions.

do the same, since the names of the clause variant must be all fresh. These choices correspond to changing by an injective renaming the variables in the result of the big-step.

Definition 24 (*Substitution associated to a transition*)
Let $\Gamma \vdash G \xrightarrow{\Lambda,\pi} \Phi \vdash G'$ be a transition. We say that the substitution θ_ρ associated to this transition is:
$\theta_\rho = \{(a(\pi(x'), \vec{y}')/x | \Lambda(x) = (a, \vec{y}), a \neq \epsilon\} \cup \{\pi(x')/x\} | \Lambda(x) = (\epsilon, \langle\rangle)\}$
for some injective renaming $\rho(x) = x', x \in \Gamma \cup \Gamma_\Lambda$.

We will now prove the correctness and the completeness of our translation.

Theorem 5 (*Correctness*)
Let $\mathcal{P}$ be a set of productions of an HSHR system as defined in definitions 9 and 10. Let P be the logic program obtained by translating the productions in $\mathcal{P}$ according to definition 23. If:
$\mathcal{P} \Vdash (\Gamma \vdash G \xrightarrow{\Lambda,\pi} \Phi \vdash G')$
then we can have in P a big-step of Synchronized Logic Programming:
$[\![\Gamma \vdash G]\!] \overset{\theta_\rho}{\Rightarrow} T$
for every ρ such that $\rho(x)$ is a fresh variable unless $x \in \Gamma \wedge \Lambda(x) = (\epsilon, \langle\rangle)$. In that case we have $\rho(x) = x$. Furthermore θ_ρ is associated to $\Gamma \vdash G \xrightarrow{\Lambda,\pi} \Phi \vdash G'$ and $T\rho^{-1} = [\![\Phi \vdash G']\!]$. Finally clauses used in the big-step are the translations of the productions used in the proof of the HSHR transition applied to the translations of the edges on which the productions were applied.

Proof
The proof is by rule induction.
See Appendix A.

Theorem 6 (*Completeness*)
Let $\mathcal{P}$ be a set of productions of an HSHR system. Let P be the logic program obtained by translating the productions in $\mathcal{P}$ according to definition 23. If we have in P a big-step of logic programming:
$[\![\Gamma \vdash G]\!] \overset{\theta}{\Rightarrow} T$
then there exist ρ, θ', Λ, π, Φ and G' such that $\theta = \theta'_\rho$ is associated to $\Gamma \vdash G \xrightarrow{\Lambda,\pi} \Phi \vdash G'$. Furthermore $T\rho^{-1} = [\![\Phi \vdash G']\!]$ and $\mathcal{P} \Vdash (\Gamma \vdash G \xrightarrow{\Lambda,\pi} \Phi \vdash G')$.

Proof
See Appendix A.

Example 8
We will here continue the example 5 by showing how that fusion computatition can be translated into a Synchronized Logic Programming computation.

$$(uxyzw)(P(x,y,z)|\overline{u}xy.Q(u,x)|uzw.R(z,w)) \rightarrow (uxy)(P(x,y,z)|Q(u,x)|R(z,w))\{x/z, y/w\}$$

Remember that a translation of the starting process is:

$$u, x, y, w, z, y_1, y_2, y_3, y_4, y_5, y_6, y_7, y_8, y_9, y_{10}, y_{11}, y_{12}, y_{13} \vdash$$
$$L_{P(x_1,x_2,x_3)}(y_1, y_2, y_3) | L_{\overline{x_1}x_2x_3.Q(x_4,x_5)}(y_4, y_5, y_6, y_7, y_8) |$$
$$L_{x_1x_2x_3.R(x_4,x_5)}(y_9, y_{10}, y_{11}, y_{12}, y_{13}) | m_4(u, y_4, y_7, y_9) | m_4(x, y_1, y_5, y_8) |$$
$$m_3(y, y_2, y_6) | m_4(z, y_3, y_{10}, y_{12}) | m_3(w, y_{11}, y_{13}) | n(u) | n(x) | n(y) | n(w) | n(z)$$

We have the following productions:

$$y_1, y_2, y_3, y_4, y_5 \vdash L_{\overline{x_1}x_2x_3.Q(x_4,x_5)}(y_1, y_2, y_3, y_4, y_5) \xrightarrow{(y_1, out_2, \langle y_2, y_3 \rangle)}$$
$$y_1, y_2, y_3, y_4, y_5 \vdash L_{Q(x_1,x_2)}(y_4, y_5) | n(y_1)$$

$$y_1, y_2, y_3, y_4, y_5 \vdash L_{x_1x_2x_3.R(x_4,x_5)}(y_1, y_2, y_3, y_4, y_5) \xrightarrow{(y_1, in_2, \langle y_2, y_3 \rangle)}$$
$$y_1, y_2, y_3, y_4, y_5 \vdash L_{R(x_1,x_2)}(y_4, y_5) | n(y_1)$$

that corresponds to the clauses (we directly write suitably renamed variants):
$L_{\overline{x_1}x_2x_3.Q(x_4,x_5)}(out_2(z_1, z_2, z_3), z_2, z_3, z_4, z_5) \leftarrow L_{Q(x_1,x_2)}(z_4, z_5) | n(z_1)$
$L_{x_1x_2x_3.R(x_4,x_5)}(in_2(w_1, w_2, w_3), w_2, w_3, w_4, w_5) \leftarrow L_{R(x_1,x_2)}(w_4, w_5) | n(w_1)$
plus the clause obtained from the auxiliary production:
$m_4(t_1, out_2(t_2, t_3, t_4), t_5, in_2(t_6, t_7, t_8)) \leftarrow m_4(t_1, t_2, t_5, t_6), m_2(t_3, t_7), m_2(t_4, t_8)$
We have the following big-step:

$$L_{P(x_1,x_2,x_3)}(y_1, y_2, y_3), L_{\overline{x_1}x_2x_3.Q(x_4,x_5)}(y_4, y_5, y_6, y_7, y_8),$$
$$L_{x_1x_2x_3.R(x_4,x_5)}(y_9, y_{10}, y_{11}, y_{12}, y_{13}),$$
$$m_4(u, y_4, y_7, y_9), m_4(x, y_1, y_5, y_8), m_3(y, y_2, y_6), m_4(z, y_3, y_{10}, y_{12}), m_3(w, y_{11}, y_{13}),$$
$$n(u), n(x), n(y), n(w), n(z)$$
$$\xrightarrow{out_2(z_1,y_5,y_6)/y_4,y_5/z_2,y_6/z_3,y_7/z_4,y_8/z_5}$$

$$L_{P(x_1,x_2,x_3)}(y_1, y_2, y_3), L_{Q(x_1,x_2)}(y_7, y_8), n(z_1),$$
$$L_{x_1x_2x_3.R(x_4,x_5)}(y_9, y_{10}, y_{11}, y_{12}, y_{13}),$$
$$m_4(u, out_2(z_1, y_5, y_6), y_7, y_9), m_4(x, y_1, y_5, y_8), m_3(y, y_2, y_6), m_4(z, y_3, y_{10}, y_{12}),$$
$$m_3(w, y_{11}, y_{13}), n(u), n(x), n(y), n(w), n(z)$$
$$\xrightarrow{u/t_1,z_1/t_2,y_5/t_3,y_6/t_4,y_7/t_5,in_2(t_6,t_7,t_8)/y_9}$$

$$L_{P(x_1,x_2,x_3)}(y_1, y_2, y_3), L_{Q(x_1,x_2)}(y_7, y_8), n(z_1),$$
$$L_{x_1x_2x_3.R(x_4,x_5)}(in_2(t_6, t_7, t_8), y_{10}, y_{11}, y_{12}, y_{13}),$$
$$m_4(u, z_1, y_7, t_6), m_2(y_5, t_7), m_2(y_6, t_8), m_4(x, y_1, y_5, y_8), m_3(y, y_2, y_6), m_4(z, y_3, y_{10}, y_{12}),$$
$$m_3(w, y_{11}, y_{13}), n(u), n(x), n(y), n(w), n(z)$$
$$\xrightarrow{t_6/w_1,y_{10}/t_7,y_{11}/t_8,y_{10}/w_2,y_{11}/w_3,y_{12}/w_4,y_{13}/w_5}$$

$$L_{P(x_1,x_2,x_3)}(y_1, y_2, y_3), L_{Q(x_1,x_2)}(y_7, y_8), n(z_1), L_{R(x_1,x_2)}(y_{12}, y_{13}), n(t_6),$$
$$m_4(u, z_1, y_7, t_6), m_2(y_5, y_{10}), m_2(y_6, y_{11}), m_4(x, y_1, y_5, y_8), m_3(y, y_2, y_6),$$
$$m_4(z, y_3, y_{10}, y_{12}), m_3(w, y_{11}, y_{13}), n(u), n(x), n(y), n(w), n(z)$$

The observable substitution of the big-step is $\{out_2(z_1, y_5, y_6)/y_4, in_2(t_6, y_{10}, y_{11})/y_9\}$. This is associated to the wanted HSHR transition with $\rho = \{z_1/y_4, t_6/y_9\}$ and by applying ρ^{-1} to the translation of the final goal we have the graph:

$$L_{P(x_1,x_2,x_3)}(y_1, y_2, y_3)|L_{Q(x_1,x_2)}(y_7, y_8)|n(y_4)|L_{R(x_1,x_2)}(y_{12}, y_{13})|n(y_9)|$$
$$m_4(u, y_4, y_7, y_9)|m_2(y_5, y_{10})|m_2(y_6, y_{11})|m_4(x, y_1, y_5, y_8)|m_3(y, y_2, y_6)|$$
$$m_4(z, y_3, y_{10}, y_{12})|m_3(w, y_{11}, y_{13})|n(u)|n(x)|n(y)|n(w)|n(z)$$

as required.

We end this section with a simple schema on the correspondence between the two models.

HSHR	SLP	HSHR	SLP
Graph	Goal	Transition	Big-step
Edge	Atomic goal	Node	Variable
Parallel comp.	And comp.	Nil	$\Box$
Production	Clause	Action	Functional s.

4.3 Using Prolog to implement Fusion Calculus

The theorems seen in the previous sections can be used for implementation purposes. As far as Synchronized Logic Programming is concerned, in Lanese (2002) a simple meta-interpreter is presented.

The idea is to use Prolog ability of dynamically changing the clause database to insert into it a set of clauses and a goal and to compute the possible synchronized computations of given length. This can be directly used to simulate HSHR transitions. In order to simulate Fusion Calculus processes we have to implement amoeboids using a bounded number of different connectors (note that m_2, m_3 and n are enough) and to implement in the meta-interpreter the condition under which productions can be applied in a single big-step. This can be easily done. Furthermore this decreases the possible choices of applicable productions and thus improves the efficiency with respect to the general case.

5 Conclusion

In this paper we have analyzed the relationships between three different formalisms, namely Fusion Calculus, HSHR and logic programming.

The correspondence between HSHR and the chosen subset of logic programming, SLP, is complete and quite natural. Thus we can consider HSHR as a "subcalculus" of (synchronized) logic programming.

The mapping between Fusion Calculus and HSHR is instead more involved because it has to deal with many important differences:

- process calculi features vs graph transformation features;
- interleaving models vs concurrent models;
- Milner synchronization vs Hoare synchronization.

Hoare synchronization was necessary since our aim was to eventually map Fusion Calculus to logic programming. If the aim is just to compare Fusion Calculus and SHR it is possible to use SHR with Milner synchronization, achieving a much simpler and complete mapping (see Lanese and Montanari 2003).

We think that the present work can suggest several interesting lines of development, dictated by the comparison of the three formalisms studied in the paper. First, our implementation of routers in terms of amoeboids is rather general and abstract, and shows that Fusion Calculus names are a rather high level concept. They abstract out the behavior of an underlying network of connections which must be open and reconfigurable. Had we chosen π-calculus instead (see a translation of π-calculus to Milner SHR in (Hirsch and Montanari 2001)), we would have noticed important differences. For instance, fusions are also considered in the semantics of *open* π-calculus by Davide Sangiorgi, but in that work not all the names can be fused: newly extruded names cannot be merged with previously generated names. This is essential for specifying nonces and session keys for secure protocols. Instead, Fusion Calculus does not provide equivalent constructs. Looking at our translation, we can conclude that logic programming does not offer this feature, either. Thus logic programming is a suitable counterpart of Fusion Calculus, but it should be properly extended for matching open π-calculus and security applications.

In a similar line a thought, we observe that we have a scope restriction operator in the Fusion Calculus, but no restriction is found in our version of HSHR. We think this omission simplifies our development, since no restriction exists in ordinary logic programming, either. However versions of SHR with restriction have been considered (Hirsch and Montanari 2001; Ferrari et al. 2001; Lanese 2002). Also (synchronized) logic programming can be smoothly extended with a restriction operator (Lanese 2002). More importantly, Fusion Calculus is equipped with an observational abstract semantics based on (hyper) bisimulation. We did not consider a similar concept for SHR or logic programming, since we considered it outside the scope of tha paper: our operational correspondence among the three formalisms is very strong and it should respect any reasonable abstract semantics. However a bisimulation semantics of SHR has been considered in (König and Montanari 2001), and an observational semantics of logic programming is discussed in (Bruni et al. 2001).

Another comment concerns concurrency. To prove the equivalence of Fusion Calculus and of its translation into HSHR we had to restricts the possible computations of the latter. On the contrary, if all computations were allowed, the same translation would yield a concurrent semantics of Fusion Calculus, that we think is worth studying. For instance in the presence of concurrent computations not all equivalent amoeboids would have the same behaviour, since some of them would allow for more parallelism than others.

Finally we would like to emphasize some practical implication of our work. In fact, logic programming is not only a model of computation, but also a well developed programming paradigm. Following the lines of our translation, implementations of languages based on Fusion Calculus and HSHR could be designed, which exploit existing ideas, algorithms and tools developed for logic programming.

References

Bruni, R., Montanari, U., and Rossi, F. 2001. An interactive semantics of logic programming. *Theory and Practice of Logic Programming 1,* 6, 647–690.

Degano, P. and Montanari, U. 1987. A model for distributed systems based on graph rewriting. *Journal of the ACM (JACM) 34,* 2, 411–449.

Ehrig, H., Kreowski, H.-J., Montanari, U., and Rozenberg, G., Eds. 1999. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol.3: Concurrency, Parallellism, and Distribution*. World Scientific.

Ferrari, G., Montanari, U., and Tuosto, E. 2001. A lts semantics of ambients via graph synchronization with mobility. In *Proc. of ICTCS'01*, S. R. D. R. Antonio Restivo and L. Roversi, Eds. LNCS, vol. 2202. Springer, 1–16.

Gardner, P. and Wischik, L. 2000. Explicit fusions. In *Mathematical Foundations of Computer Science*. 373–382.

Gardner, P. and Wischik, L. 2004. Strong bisimulation for the explicit fusion calculus. In *Foundations of Software Science and Computation Structures*. To appear.

Hirsch, D., Inverardi, P., and Montanari, U. 2000. Reconfiguration of software architecture styles with name mobility. In *Proceedings of the 4th International Conference on Coordination Languages and Models, Coordination 2000*. Lecture Notes in Computer Science, vol. 1906.

Hirsch, D. and Montanari, U. 2001. Synchronized hyperedge replacement with name mobility. In *Proc. of CONCUR'01*, S. Verlag, Ed. LNCS.

König, B. and Montanari, U. 2001. Observational equivalence for synchronized graph rewriting. In *Proc. TACS'01*, S. Verlag, Ed. LNCS, vol. 2215.

Lanese, I. 2002. Process synchronization in distributed systems via Horn clauses. M.S. thesis, University of Pisa, Computer Science Department. Downloadable from `http://www.di.unipi.it/~lanese/tesi.ps`.

Lanese, I. and Montanari, U. 2002. Software architectures, global computing and graph transformation via logic programming. In *Proc SBES'2002 - 16th Brazilian Symposium on Software Engineering*, L. Ribeiro, Ed. Anais, 11–35.

Lanese, I. and Montanari, U. 2003. A graphical fusion calculus. Proceedings of CoMeta Final Workshop, Electronic Notes in Computer Science, to appear.

Milner, R., Parrow, J., and Walker, D. 1992. A calculus of mobile processes. *Inform. and Comput. 100*, 1–77.

Parrow, J. and Victor, B. 1998. The fusion calculus: Expressiveness and symmetry in mobile processes. In *Proceedings of LICS '98*. IEEE, Computer Society Press.

Victor, B. 1998. The fusion calculus: Expressiveness and symmetry in mobile processes. Ph.D. thesis, Dept. of Computer Systems, Uppsala University, Sweden.

Appendix A Proofs

We have here the proofs that are missing in the main part and some lemmas used in these proofs. Lemmas are just before the proofs that use them.

Lemma 3
Given a set of amoeboids for σ and a substitutive effect θ of a fusion $\phi = \{x_i = y_i\}_{i=1\ldots|\vec{x}|}$ then $|_{i=1\ldots|\vec{x}|} m_2(x_i, y_i)|[\![\sigma]\!]$ is a set of amoeboids for $\sigma\theta$.

Proof
Since we are working up-to injective renamings we only have to prove that two names are connected by $|_{i=1\ldots|\vec{x}|} m_2(x_i, y_i)|[\![\sigma]\!]$ iff they are merged by $\sigma\theta$. By definition two names are connected by $[\![\sigma]\!]$ iff they are merged by σ. Assume that two names x and y are merged by $\sigma\theta$. Then their images along σ are merged by θ. This means that we have in ϕ a chain of equalities from $\sigma(x)$ to $\sigma(y)$. Thus we have m_2 edges connecting the amoeboids of $\sigma(x)$ and $\sigma(y)$, thus x and y are connected. Assume now that x and y are connected. Then there exists a chain of m_2 edges connecting the amoeboids for $\sigma(x)$ and $\sigma(y)$. Thus θ merges $\sigma(x)$ and $\sigma(y)$ as required. □

Proof of theorem 2(Sketch)
The proof is by rule induction on the reduction semantics.
Let us consider the reduction rule:

$$(\vec{z})(R|(\cdots + u\vec{x}.P)|(\overline{u}\vec{y}.Q + \ldots)) \rightarrow (\vec{z})(R|P|Q)\sigma$$

where $|\vec{x}| = |\vec{y}|$ and σ is a substitutive effect of $\{\vec{x} = \vec{y}\}$. We see in the LHS two sequential processes $(\cdots + u\vec{x}.P)$ and $(\overline{u}\vec{y}.Q + \ldots)$. When translating, these two processes are modeled by two edges connected to distinct nodes. The nodes corresponding to u in the two processes are connected by an amoeboid. Thus we can have a transition synchronizing the two productions that correspond to the prefixes $u\vec{x}.P$ and $\overline{u}\vec{y}.Q$ and the productions for the amoeboid for u. A complete transition can be built using (idle) rules for all the non rewritten edges and using (new) rules to deal with isolated nodes.
Note that we have *id* as fusion since we have no merges in the productions and when we merge two nodes because of a synchronization at least one of them is a new node (amoeboids expose only new nodes), thus we have no merges between old nodes.
We need to prove that the result of the transition is the translation of the wanted fusion process up-to injective renamings and up-to the structure of the amoeboids. First of all note that in the final graph each node is connected to two edges. In fact the only connections that are not preserved by productions are the ones between nodes whose references are exposed, namely $\vec{x}$ and $\vec{y}$, and the exposing process edges but these nodes are connected to the new m_2 edges created by the auxiliary productions.
The correctness of the process part of the graph is trivial. The correctness of the amoeboid part follows from lemma 3.

$$(\vec{z})(R|(\cdots + \phi.P)) \rightarrow (\vec{z})(R|P)\sigma$$

where σ is a substitutive effect of ϕ.
In this case we rewrite using a production just one edge, which is the edge that

corresponds to the sequential process $(\cdots + \phi.P)$. In this case the effect of the fusion is obtained by applying its substitutive effect. In the substitutive effect we have a binding x/y for each equality in ϕ (remember that ϕ is linear). Thus the effect of the transition is of substituting the translation of $(\cdots + \phi.P)$ with the translation of P, merging nodes corresponding to names merged by ϕ and closing the remaining unused nodes. Note that we always merge nodes connected to exactly one amoeboid, thus the final effect is to connect pairs of amoeboids.

$$\frac{P \equiv P', P' \to Q', Q' \equiv Q}{P \to Q}$$

Equivalent agents are converted into the same representative (up-to α-conversion) before being translated, thus the translation of P and P' are equal up-to injective renamings. Similarly for the translation of Q and Q', thus the thesis follows. □

Lemma 4
Given a graph $\Gamma \vdash G$ and one or zero productions for each edge in G let:
$\Gamma \vdash G \xrightarrow{\Lambda_1,\pi_1} \Phi_1 \vdash G'_1$
$\Gamma \vdash G \xrightarrow{\Lambda_2,\pi_2} \Phi_2 \vdash G'_2$
be two transitions obtained by applying the chosen production for each edge (and using the (idle) rule if no production is chosen). Then there exists an injective renaming σ such that:

- $\Lambda_1(x) = \sigma(\Lambda_2(x))$ if x is not an isolated node;
- $\pi_1 = \sigma\pi_2$;
- $\Phi_1 = \sigma(\Phi_2)$;
- $G'_1 = \sigma(G'_2)$.

Sketch
The proof is a simple rule induction if one proves that derivations can be done in a standard way, namely by applying to the axioms first rules (par), then rule (merge) and finally rules (new). This can be proved by showing that one can exchange the order of rules and that one can substitute two applications of (merge) with substitutions σ and σ' with just one application with substitution $\sigma' \cdot \sigma$. □

Proof of theorem 3(Sketch)
Let us first consider the case of two productions for communication actions.
In order to apply them we need two sequential process edges to be rewritten. Each production needs to be synchronized with at least another one. Since process edges are connected only through amoeboids we can have a synchronization only if the two actions done by process edges are complementary actions. Furthermore they have to be done on the same amoeboid, that is on two names merged by the substitution σ corresponding to the amoeboids.
Thus P can be decomposed in the form $(\vec{x})P'\sigma$ where $P' = P'_1|P'_2|Q'$ with P'_1 and P'_2 sequential processes which are translated into the rewritten edges.
We must have $P'_1 = \cdots + u_1\vec{x}.P''_1$ and $P'_2 = \overline{u_2}\vec{y}.P''_2 + \ldots$ (or swapped) with $|\vec{x}| = |\vec{y}|$.
Furthermore σ merges u_1 and u_2 thus we have a transition $P \to P'$ that corresponds

to the synchronized execution of the two prefixes.
From lemma 4 we know that the result of a transition is determined up-to injective renamings (and actions on isolated nodes) by the starting graph and the productions chosen. Thus the transition that corresponds to $P \to P'$ for theorem 2 is equal up-to injective renamings to $[\![P]\!]_{\vec{v},\vec{w}} \xrightarrow{\Lambda,\pi} \Gamma' \vdash G$.
The thesis follows.
The other case is analogous. □

Lemma 5
Let:
$G_0 \xrightarrow{\theta'_1} \cdots \xrightarrow{\theta'_n} G_n$ be a computation.
For each i, j such that $1 \leq i < j \leq n$ we have $\mathrm{dom}(\theta'_i) \cap \mathrm{n}(\theta'_j) = \emptyset$.

Proof
$G_i = G'_{i-1}\theta_i$
where G'_{i-1} is obtained from G_{i-1} by substituting the body of the clause variant used to the unified atom and θ_i is the actual mgu.

Because θ_i is idempotent we have:
$\mathrm{dom}(\theta_i) \cap \mathrm{n}(G_i) = \emptyset$.
We can observe that the variables in $\mathrm{dom}(\theta_i)$ are not reintroduced in the computation because at each step all the names come from the previous goal or from the clause variant (because we use relevant mgus). In the first case they could not be in $\mathrm{dom}(\theta_i)$ by inductive hypothesis, in the second case because they are new. □

Lemma 6
Let $A_1, \ldots, A_n$ be a goal.
We want to build a big-step where the clause unified with A_i is $H_i \leftarrow B_i$ (if we want to replace A_i, otherwise we define $B_i = A_i$). As a notational convention we use $x_{i,1}, \ldots, x_{i,n_i}$ to denote the arguments of A_i and $a_{i,j}(x'_{i,j}, \vec{y'_{i,j}})$ to denote the jth argument of H_i if it is a complex term and $x'_{i,j}$ if it is a variable.
Let θ_r be the mgu of the following set of equations: $\{x'_{i,j} = x'_{p,q}, y'_{i,j,k} = y'_{p,q,k} | x_{i,j} = x_{p,q}\} \cup \{x_{i,j} = x'_{i,j} | \text{the jth argument of } H_i \text{ is } x'_{i,j}\}$. We will denote $\theta_r(x)$ with $[x]$.
We will have a big-step of the form:
$A_1, \ldots, A_n \overset{\theta}{\Rightarrow} G_1, \ldots, G_n$
iff
$x_{i,j} = x_{p,q} \Rightarrow a_{i,j} = a_{p,q}$
Furthermore we have:
$\theta = \{a_{i,j}([x'_{i,j}], [\vec{y'_{i,j}}])/x_{i,j} | a_{i,j} \text{ is defined}\} \cup \{[v_k]/v_k | (v_k = x'_{i,j} \vee v_k = y'_{i,j,k} \vee (v_k = x_{i,j} \wedge a_{i,j} \text{ is undefined }) \wedge [v_k] \neq v_k\}$
$G_1, \ldots, G_n = (B_1, \ldots, B_n)\theta_r$.
Note that the big-step is determined (up-to injective renamings) by the choice of the clauses and of the atoms they are applied to.

Proof
We will prove a more general result by induction on the number of "considered" atoms.

We consider partial computations where only considered atoms are expanded.
We will prove that a computation of the form $A_1, \ldots, A_n \stackrel{\theta}{\rightarrow}^* G_1, \ldots, G_n$ where we substitute only atoms in the starting goal and where the atoms generated by considered atoms do not contain function symbols exists iff:
$x_{i,j} = x_{p,q} \Rightarrow a_{i,j} = a_{p,q}$ if $x_{i,j}$ and $x_{p,q}$ are in considered atoms and that furthermore:

- $\theta = \{a_{i,j}([x'_{i,j}]', [\vec{y'_{i,j}}]')/x_{i,j} | a_{i,j}$ is defined and in a considered atom$\} \cup \{[v_k]'/v_k | (v_k = x'_{i,j} \vee v_k = y'_{i,j,k} \vee (v_k = x_{i,j} \wedge a_{i,j}$ is undefined $) \wedge [v_k]' \neq v_k\}$;
- $G_1, \ldots, G_n = (B_1, \ldots, B_k)\theta'_r, (A_{k+1}, \ldots, A_n)\theta$ where the first k atoms are considered;

where we use as θ'_r the relation obtained using only equalities between variables in considered atoms and where $\theta'_r(x) = [x]$.

Note that if all the atoms are considered we obtain the thesis.
1) We have the following transition:
$A_1, \ldots, A_n \stackrel{\theta_1}{\longrightarrow} (B_1, A_2, \ldots, A_n)\theta_1$
where θ_1 is an mgu of $\{A_1 = H_1\}$. This transition exists iff θ_1 exists.
We have:
$\theta_1 = \text{mgu}(\{a_{1,j}(x'_{1,j}, \vec{y}'_{1,j}) = x_{1,j} | a_{1,j}$ is defined$\} \cup \{x'_{1,j} = x_{1,j} | a_{1,j}$ is undefined$\})$.
Note that if $x_{1,j} = x_{1,q} \not\Rightarrow a_{1,j} = a_{1,q}$ then the mgu does not exist (if $a_{i,j}$ and $a_{1,q}$ are both defined, otherwise they have to be both undefined. In fact if just one of them is defined a functional symbol will remain in the considered part against the hypothesis).
If the condition is satisfied we have:

$$\begin{aligned}\theta_1 = \text{mgu}(\{a_{1,j}(x'_{1,j}, \vec{y}'_{1,j}) = x_{1,j} | a_{1,j} \text{ is defined}\} \cup \\ \cup \{a_{1,j}(x'_{1,j}, \vec{y}'_{1,j}) = a_{1,p}(x'_{1,p}, \vec{y}'_{1,p}) | x_{1,j} = x_{1,p}\} \cup \{x'_{1,j} = x_{1,j} | a_{1,j} \text{ is undefined}\}) = \\ = \text{mgu}(\{a_{1,j}(x'_{1,j}, \vec{y}'_{1,j}) = x_{1,j} | a_{1,j} \text{ is defined}\} \cup \\ \cup \{x'_{1,j} = x'_{1,q}, \vec{y}'_{1,j} = \vec{y}'_{1,q} | x_{1,j} = x_{1,q}\} \cup \{x'_{1,j} = x_{1,j} | a_{1,j} \text{ is undefined}\})\end{aligned}$$

Note that the last part is exactly the set of equations that corresponds to θ'_r, thus θ'_r is its mgu.
We thus have:
$\theta_1 = \text{mgu}(\{a_{1,j}(x'_{1,j}, \vec{y}'_{1,j}) = x_{1,j} | a_{1,j}$ is defined$\} \cup \{[v_k]'/v_k | [v_k]' \neq v_k\})$.
To have the real mgu we just need to apply θ'_r to the equations in the first part (note that $x_{1,j}$ is unified with some other variable only if $a_{1,j}$ is undefined thus the domain variables of the first part are not changed).
This proves the first part since this mgu exists and the second one since it has the wanted form.
The third part follows from the observation that $\theta_1|_{\text{n}(B_1)} = \theta'_r|_{\text{n}(B_1)}$.
Inductive step)
Assume that $A_1, \ldots, A_n \stackrel{\theta_g}{\longrightarrow}^* G_1, \ldots, G_n$ is a logic computation where we substitute

only atoms in the starting goal and where the atoms generated by considered atoms do not contain function symbols and that atoms $A_1, \ldots, A_{k+1}$ are considered.
Let us take the computation where only the first k atoms are considered.

By inductive hypothesis we have that this part of the computation exists iff:
$x_{i,j} = x_{p,q} \Rightarrow a_{i,j} = a_{p,q}$ if $x_{i,j}$ and $x_{p,q}$ are considered
and that:

- $\theta = \{a_{i,j}([x'_{i,j}], [\vec{y'_{i,j}}])/x_{i,j} | a_{i,j}$ is defined and in a considered atom$\} \cup \{[v_k]/v_k | (v_k = x'_{i,j} \vee v_k = y'_{i,j,k} \vee (v_k = x_{i,j} \wedge a_{i,j}$ is undefined $) \wedge [v_k] \neq v_k\}$;

- $G_1, \ldots, G_n = (B_1, \ldots, B_k)\theta_r, (A_{k+1}, \ldots, A_n)\theta$.

We will now consider the atom A_{k+1}. If it is not substituted then the thesis follows trivially. Let us consider the case in which it is substituted.

We will have a small step of the form:
$(G_1, \ldots, G_k, A_{k+1}, \ldots, A_n)\theta \xrightarrow{\theta'} (\ G_1, \ldots, G_{k+1}, A_{k+2}, \ldots, A_n)\theta\theta'$
where $\theta' = \mathrm{mgu}(\{A_{k+1}\theta = H_{k+1}\})$ (note that we can assume that θ is also applied to G_{k+1} since $\mathrm{dom}(\theta) \cap \mathrm{n}(G_{k+1}) = \emptyset$).
We have:

$$\begin{aligned}\theta' = \mathrm{mgu}(\{A_{k+1}\theta = H_{k+1}\}) = \\ = \mathrm{mgu}(\{(a_{k+1,j}(x'_{k+1,j}, \vec{y'}_{k+1,j}) = x_{k+1,j})\theta | a_{k+1,j} \text{ is defined}\} \cup \\ \cup \{(x'_{k+1,j} = x_{k+1,j})\theta | a_{k+1,j} \text{ is undefined}\})\end{aligned}$$

For each binding we must consider two cases: either the variable $x_{k+1,j}$ appears in already considered atoms or it does not. In the second case θ is the identity on that variable. In the first case if $a_{k+1,j}$ is defined then the mgu exists iff we have $a_{k+1,j} = a_{p,q}$ where $a_{p,q}$ is the functional symbol in the binding for $x_{k+1,j}$ in θ. Note that if $a_{p,q}$ is undefined then also $a_{k+1,j}$ must be undefined otherwise the functional symbol remains in the final goal.
Thus we will have:

$$\begin{aligned}\theta' = \mathrm{mgu}(\{A_{k+1}\theta = H_{k+1}\}) = \\ = \mathrm{mgu}(\{a_{k+1,j}(x'_{k+1,j}, \vec{y'}_{k+1,j}) = a_{p,q}([x'_{p,q}], [\vec{y'_{p,q}}])| \\ a_{k+1,j} \text{ is defined and } x_{k+1,j} = x_{p,q} \text{ appears in an already considered atom}\} \cup \\ \cup \{x'_{k+1,j} = [x_{k+1,j}] | a_{k+1,j} \text{ is undefined}\} \cup \\ \cup \{a_{k+1,j}(x'_{k+1,j}, \vec{y'}_{k+1,j}) = x_{k+1,j}| \\ a_{k+1,j} \text{ is defined and } x_{k+1,j} \text{ does not appear in an already considered atom }\} = \\ = \mathrm{mgu}(\{x'_{k+1,j} = [x'_{p,q}], \vec{y'}_{k+1,j} = \vec{[y'_{p,q}]}| \\ a_{k+1,j} \text{ is defined and } x_{k+1,j} = x_{p,q} \text{ appears in an already considered atom}\} \cup \\ \cup \{x'_{k+1,j} = [x_{k+1,j}] | a_{k+1,j} \text{ is undefined}\} \cup \\ \cup \{a_{k+1,j}(x'_{k+1,j}, \vec{y'}_{k+1,j}) = x_{k+1,j}| \\ a_{k+1,j} \text{ is defined and } x_{k+1,j} \text{ does not appear in an already considered atom }\}\end{aligned}$$

where we use $[-]$ to denote the representative of the equivalence class according to θ. We can reorder this substitution into

$$\begin{aligned}&\mathrm{mgu}(\{a_{k+1,j}(x'_{k+1,j}, \vec{y}'_{k+1,j}) = x_{k+1,j} \mid \\ &a_{k+1,j} \text{ is defined and } x_{k+1,j} \text{ does not appear in an already considered atom } \} \cup \\ &\qquad \cup \{x'_{k+1,j} = [x'_{p,q}], \vec{y}'_{k+1,j} = \vec{[}y'_{p,q}]\mid \\ &a_{k+1,j} \text{ is defined and } x_{k+1,j} = x_{p,q} \text{ appears in an already considered atom}\} \cup \\ &\qquad \cup \{x'_{k+1,j} = [x_{k+1,j}] \mid a_{k+1,j} \text{ is undefined}\}\end{aligned}$$

Note that the second part is a renaming that does not involve variables in the domain of the first part. Thus we can define new equivalence classes $[-]'$ according to this substitution θ'_r and put the substitution in the resolved form:

$$\begin{aligned}&\mathrm{mgu}(\{a_{k+1,j}([x'_{k+1,j}]', [\vec{y}'_{k+1,j}]') = x_{k+1,j} \mid \\ &a_{k+1,j} \text{ is defined and } x_{k+1,j} \text{ does not appear in an already considered atom } \} \cup \\ &\qquad \cup \{[v_k]'/v_k \mid [v_k]' \neq v_k\})\end{aligned}$$

Thus the mgu exists and the first part of the thesis is proved.

Let us consider the substitution $\theta_g = \theta\theta'$. Note that the renaming part of θ' substitutes variables according to the equivalence between variables in the H_{k+1} and representatives of the corresponding variables in A_{k+1} thus the composed substitution θ''_r maps each variable to the representative of the equivalence class that is defined by the union of the two sets of equations as required. Furthermore bindings with complex terms (the first part of the substitution) are disjoint and thus the union is made. Bindings coming from θ' have already the wanted representatives in the image, while to bindings in θ the renaming is applied mapping variables into their representatives. Thus θ_g has the wanted form with respect to the equivalence classes determined by all the equivalence on variables.
We have $(G_1, \ldots, G_{k+1}, A_{k+2}, \ldots, A_n)\theta_g = (G_1, \ldots, G_{k+1})\theta''_r, (A_{k+2}, \ldots, A_n)\theta_g$ as required since $(\mathrm{dom}(\theta_g) \setminus \mathrm{dom}(\theta''_r)) \cap \mathrm{n}(G_1, \ldots, G_{k+1}) = \emptyset$.
Note that this result does not depend on the order of application of clauses and that after having chosen which clauses to apply and to which atoms it is deterministic up-to an injective renaming (which depends on the choice of names for new variables and on the choice of representatives for the equivalence classes). □

Proof of theorem 5
The proof is by rule induction.
Axioms)
Assume we have a production:
$x_1, \ldots, x_n \vdash s(x_1, \ldots, x_n) \xrightarrow{\Lambda,\pi} \Phi \vdash G$
where:
$\Lambda = \{(x_i, a_i, \vec{y}_i)\}_{i=1\ldots n}$
and $\pi : \{x_1, \ldots, x_n\} \to \{x_1, \ldots, x_n\}$ is an idempotent substitution.

Then we have in P a clause:
$s(a_1(\pi(x_i), \vec{y}_1), \ldots, a_n(\pi(x_n), \vec{y}_n)) : -[\![\Phi \vdash G]\!]$

where we have $\pi(x_i)$ instead of $a_i(\pi(x_i), \vec{y}_i)$ if $a_i = \epsilon$.
We can have an applyable variant of this clause for each injective renaming ρ that maps each variable to a fresh one.
Let us consider the goal:
$[\![s(x_1, \ldots, x_n)]\!] = s(x_1, \ldots, x_n)$
It unifies with the clause variant:
$s(a_1(\pi(x_i), \vec{y}_1), \ldots, a_n(\pi(x_n), \vec{y}_n))\rho : -[\![\Phi \vdash G]\!]\rho$
with mgu θ_ρ:
$\theta_\rho = \{a_i(\pi(x_i)\rho, \vec{y}_i\rho)/x_i | a_i \neq \epsilon\} \cup \{\pi(x_i)\rho/x_i | a_i = \epsilon\}$.
Note that $\theta_\rho = \theta_\rho|_{\mathrm{n}(s(x_1,\ldots,x_n))}$.
We can see that θ_ρ is associated to $x_1, \ldots, x_n \vdash s(x_1, \ldots, x_n) \xrightarrow{\Lambda,\pi} \Phi \vdash G$ as required.
Observe also that the result of the computation is:
$T = [\![\Phi \vdash G]\!]\rho\theta_\rho = [\![\Phi \vdash G]\!]\rho$
because $\mathrm{n}(G\rho) \cap \mathrm{dom}(\theta_\rho) = \emptyset$.
So because π is the identity on all the variables in G we have:
$T\rho^{-1} = [\![\Phi \vdash G]\!]\rho\rho^{-1} = [\![\Phi \vdash G]\!]$
as required.
To deal with the missing possibilities for ρ with respect to the theorem note that bindings in the last of θ_ρ can also be resolved also as $x_i/\pi(x_i)\rho$. In that case we have no binding for x_i in θ_ρ. This is equivalent to defining $x_i\rho = x_i$, what covers the missing cases for ρ.

Note finally that we used as clause the translation of the production and that we applied the clause to the translation of the rewritten edge.
Rule (par))

$$\frac{\Gamma \vdash G_1 \xrightarrow{\Lambda,\pi} \Phi \vdash G_2 \qquad \Gamma' \vdash G'_1 \xrightarrow{\Lambda',\pi'} \Phi' \vdash G'_2}{\Gamma, \Gamma' \vdash G_1|G'_1 \xrightarrow{\Lambda\cup\Lambda',\pi\cup\pi'} \Phi, \Phi' \vdash G_2|G'_2}$$

where $(\Gamma \cup \Phi) \cap (\Gamma' \cup \Phi') = \emptyset$.
By inductive hypothesis we have:

- $[\![\Gamma \vdash G_1]\!] \overset{\theta_\rho}{\Rightarrow} T$
  where θ_ρ is associated to $\Gamma \vdash G_1 \xrightarrow{\Lambda,\pi} \Phi \vdash G_2$ and $T\rho^{-1} = [\![\Phi \vdash G_2]\!]$;
- $[\![\Gamma' \vdash G'_1]\!] \overset{\theta'_{\rho'}}{\Rightarrow} T'$
  where $\theta'_{\rho'}$ is associated to $\Gamma' \vdash G'_1 \xrightarrow{\Lambda',\pi'} \Phi' \vdash G'_2$ and $T'\rho'^{-1} = [\![\Phi' \vdash G'_2]\!]$.

  In both cases by inductive hypothesis we used as clauses the translations of the productions used in the proof of the HSHR rewriting applied to the translations of the edges on which the productions were applied.

Thanks to lemma 1 we can assume that the sets of variables used in the two computations are disjoint. Thanks to lemma 6 we know that these big-steps exist iff $x_{i,j} = x_{p,q} \Rightarrow a_{i,j} = a_{p,q}$. Since the used variable sets are disjoint the same condition guarentees the existence of a big-step of the form:

$[\![\Gamma, \Gamma' \vdash G_1|G_1']\!] \xrightarrow{\theta_\rho\theta'_{\rho'}}{}^{*} T, T'$
where we used as clauses the unions of the clauses used in the two smaller computations, applied to the same predicates.

We have that $\theta_\rho\theta'_{\rho'}$ is associated to:
$\Gamma, \Gamma' \vdash G_1|G_1' \xrightarrow{\Lambda\cup\Lambda', \pi\cup\pi'} \Phi, \Phi' \vdash G_2|G_2'$.
We also have $\theta_\rho\theta'_{\rho'} = (\theta\theta')_{\rho\rho'}$ and thus:
$(T, T')\rho^{-1}\rho'^{-1} = T\rho^{-1}, T'\rho'^{-1} = [\![\Phi \vdash G_2]\!], [\![\Phi' \vdash G_2']\!] = [\![\Phi, \Phi' \vdash G_2|G_2']\!]$
Rule (merge))

$$\frac{\Gamma \vdash G_1 \xrightarrow{\Lambda,\pi} \Phi \vdash G_2}{\sigma(\Gamma) \vdash \sigma(G_1) \xrightarrow{\Lambda',\pi'} \Phi' \vdash \rho_g(\sigma(G_2))}$$

where $\sigma : \Gamma \to \Gamma$ is an idempotent substitution and:
$\sigma(x) = \sigma(y) \wedge x \neq y \Rightarrow \Lambda(x) = (a, \vec{v}) \wedge \Lambda(y) = (a, \vec{w})$
$\rho_g = \mathrm{mgu}(\{\sigma(\vec{v}) = \sigma(\vec{w}) | \sigma(x) = \sigma(y) \wedge \Lambda(x) = (a, \vec{v}) \wedge \Lambda(y) = (a, \vec{w})\} \cup \{\sigma(x) = \sigma(y) | \pi(x) = \pi(y)\})$
$\Lambda'(\sigma(z)) = \rho_g(\sigma(\Lambda(z)))$
$\pi' = \rho_g|_{\sigma(\Gamma)}$
For inductive hypothesis we have:
$[\![\Gamma \vdash G_1]\!] \stackrel{\theta_\rho}{\Rightarrow} T$
where θ_ρ is associated to $\Gamma \vdash G_1 \xrightarrow{\Lambda,\pi} \Phi \vdash G_2$ and $T\rho^{-1} = [\![\Phi \vdash G_2]\!]$ and we used as clauses the translations of the productions used in the proof of the HSHR transition applied to the translations of the edges on which the productions were applied.

Thanks to lemma 6 we have that this computation exists iff:
$x_{i,j} = x_{p,q} \Rightarrow a_{i,j} = a_{p,q}$
and that:

- $\theta_r = \mathrm{mgu}(\{x'_{i,j} = x'_{p,q}, y'_{i,j,k} = y'_{p,q,k} | x_{i,j} = x_{p,q}\} \cup$
  $\{x_{i,j} = x'_{i,j} | \text{the jth argument of } H_i \text{ is } x'_{i,j}\})$

- $\theta = \{a_{i,j}([x'_{i,j}], [\vec{y'_{i,j}}])/x_{i,j} | a_{i,j} \text{ is defined}\} \cup \{[v_k]/v_k | (v_k = x'_{i,j} \vee v_k = y'_{i,j,k} \vee (v_k = x_{i,j} \wedge a_{i,j} \text{ is undefined }) \wedge [v_k] \neq v_k\}$

- $T = (B_1, \ldots, B_n)\theta_r$

  where $[\![\Gamma \vdash G_1]\!] = A_1, \ldots, A_n$ and we used the name conventions defined in lemma 6.

  We have:
  $\sigma([\![\Gamma \vdash G_1]\!]) = [\![\sigma(\Gamma) \vdash \sigma(G_1)]\!]$
  We want to find a big-step that uses the same clauses of the previous one applied to the same atoms, but starting from this new goal. We will use an overline to denote the components of the new big-step.
  Thanks to lemma 6 such a big-step exists iff:
  $\sigma(x_{i,j}) = \sigma(x_{p,q}) \Rightarrow a_{i,j} = a_{p,q}$
  but if $x_{i,j} = x_{p,q}$ this has already been proved, otherwise this is guarenteed by the applicability conditions of the rule and the definition of associated transition.

We have to prove that $\overline{\theta}_{\overline{\rho}}$ is associated to:
$\sigma(\Gamma) \vdash \sigma(G_1) \xrightarrow{\Lambda',\pi'} \Phi' \vdash \rho_g(\sigma(G_2))$.
We have:
$\overline{\theta}_{\overline{\rho}} = \{a_{i,j}([x'_{i,j}]', [\vec{y'_{i,j}}]')/\sigma(x_{i,j}) | a_{i,j} \text{ is defined}\} \cup \{[v_k]'/v_k | (v_k = x'_{i,j} \vee v_k = y'_{i,j,k} \vee (v_k = x_{i,j} \wedge a_{i,j} \text{ is undefined }) \wedge [v_k]' \neq v_k\}$
where $[-]'$ maps each name to the representative of the equivalence class given by:

$$\{x'_{i,j} = x'_{p,q}, y'_{i,j,k} = y'_{p,q,k} | \sigma(x_{i,j}) = \sigma(x_{p,q})\} \cup \\ \cup \{\sigma(x_{i,j}) = x'_{i,j} | \text{the jth argument of } H_i \text{ is } x'_{i,j}\}$$

Note that $\sigma(x_{i,j})$ is never unified with a primed variable unless $a_{i,j}$ is undefined.
We need to prove that $[x'_{i,j}]' = [x'_{p,q}]'$ iff $\pi'(\sigma(x_{i,j})) = \pi'(\sigma(x_{p,q}))$ and that a similar condition holds for variables $y_{i,j,k}$. This can be proved by showing that the mgu computed in the two settings, when restricted to simple names (for the logic programming environment), are analogous up-to ρ.
This proves that $\overline{\theta}_{\overline{\rho}}$ is associated to $\sigma(\Gamma) \vdash \sigma(G_1) \xrightarrow{\Lambda',\pi'} \Phi' \vdash \rho_g(\sigma(G_2))$ as required.

By hypothesis and using lemma 6 we have:
$[\![\Phi \vdash G_2]\!] = (B_1, \ldots, B_n)\theta_r)\rho^{-1}$
The result of the new computation is:
$(B_1, \ldots, B_n)\overline{\theta_r}$
Thanks to the properties of $\overline{\rho}$ we have:
$((B_1, \ldots, B_n)\overline{\theta_r})\overline{\rho}^{-1} = \rho_g(\sigma((B_1, \ldots, B_n)\theta_r)\overline{\rho}^{-1}) = \rho_g(\sigma([\![\Phi \vdash G_2]\!])) = [\![\Phi' \vdash \rho_g(\sigma(G_2))]\!]$.
Rule $(idle)$)

$$\Gamma \vdash G \xrightarrow{\Lambda_\epsilon, id} \Gamma \vdash G$$

The proof is analogous to the proof for axioms, using as clause corresponding to the production the translation of the (idle) rule.
Rule (new))

$$\frac{\Gamma \vdash G_1 \xrightarrow{\Lambda,\pi} \Phi \vdash G_2}{\Gamma, x \vdash G_1 \xrightarrow{\Lambda \uplus \{(x,a,\vec{y})\}} \Phi' \vdash G_2}$$

For inductive hypothesis we have in P the following big-step of Synchronized Logic Programming:
$[\![\Gamma \vdash G_1]\!] \xrightarrow{\theta_\rho}{}^{*} T$
where θ_ρ is associated to:
$\Gamma \vdash G_1 \xrightarrow{\Lambda,\pi} \Phi \vdash G_2$
and $T\rho^{-1} = [\![\Phi \vdash G_2]\!]$.
But because $x \notin \mathrm{n}(G_1)$ we have that θ_ρ is also associated to:
$\Gamma, x \vdash G_1 \xrightarrow{\Lambda \uplus \{(x,a,\vec{y})\},\pi} \Phi' \vdash G_2$
The thesis follows. □

Proof of theorem 6

Our translation $[\![\Gamma \vdash G]\!]$ can be written in the form $A_1, \ldots, A_n$ where the $A_i s$ are translations of single edges.
We want to associate an HSHR rewriting to each A_i.
For edges associated to goals that are rewritten in $[\![\Gamma \vdash G]\!] \stackrel{\theta}{\Rightarrow} T$ we use an instance of the axiom that corresponds to the clause used to rewrite it, otherwise the rule obtained from rule (idle) applied to that edge and to nodes that are all distinct.
Let $T_1, \ldots, T_n$ be the heads of these rules. We choose the names in the following way: for each first occurrence of a name in $A_1, \ldots, A_n$ we use the same name for the node in the corresponding position, we use new names for all other occurrences. Note that there exists a substitution σ such that $\sigma(T_1, \ldots, T_n) = A_1, \ldots, A_n$ and that σ is idempotent. Note also that for each i all names in T_i are distinct.
For each i we can choose an instance R_i of the associated rule with head T_i such that for each $i, j, i \neq j$ we have $\mathrm{n}(R_i) \cap \mathrm{n}(R_j) = \emptyset$.
As notation we use:
$R_i = \Gamma_i \vdash G_i \xrightarrow{\Lambda_i, \pi_i} \Phi_i \vdash G'_i$
Since all the rules have a disjoint set of names we can apply $n-1$ times rule (par) in order to have:
$\bigcup_i \Gamma_i \vdash |_i G_i \xrightarrow{\bigcup_i \Lambda_i, \bigcup_i \pi_i} \bigcup_i \Phi_i \vdash |_i G'_i$
Now we want to apply rule (merge) with substitution σ. We can do it since σ is idempotent.
We have to verify that $\sigma(x) = \sigma(y) \wedge x \neq y \Rightarrow (\bigcup_i \Lambda_i)(x) = (a, \vec{v}) \wedge (\bigcup_i \Lambda_i)(x) = (a, \vec{w})$. This happens thanks to lemma 6.
Thus we obtain a rule of the form:
$\mathrm{n}(A_1, \ldots, A_n) \vdash A_1, \ldots, A_n \xrightarrow{\Lambda, \pi} \Phi \vdash G'$
Thanks to theorem 5 we can have in P the following big-step of Synchronized Logic Programming:
$[\![\mathrm{n}(A_1, \ldots, A_n) \vdash A_1, \ldots, A_n]\!] \stackrel{\theta'_\rho}{\Rightarrow} T'$
for every ρ that satisfies the freshness conditions. Furthermore θ'_ρ is associated to $\mathrm{n}(A_1, \ldots, A_n) \vdash A_1, \ldots, A_n \xrightarrow{\Lambda, \pi} \Phi \vdash G'$ and $T'\rho^{-1} = [\![\Phi \vdash G']\!]$. Finally we used as clauses the translations of the productions used in the proof of the HSHR rewriting. Note that $[\![\mathrm{n}(A_1, \ldots, A_n) \vdash A_1, \ldots, A_n]\!] = [\![\Gamma \vdash G]\!]$. Since the result of a big-step is determined up-to injective renaming by the starting goal and the used clauses (see lemma 6) then we must have $\theta = \theta'_\rho \rho'$ and $T = T'\rho'$ for some injective renaming ρ'. Note that $\rho\rho'^{-1}$ satisfies the freshness conditions (since variables generated in logic programming are always fresh) and we also have $\theta = \theta''_{\rho\rho'^{-1}}$.
Thus we have that θ is associated to $\Gamma \vdash G \xrightarrow{\Lambda, \pi} \Phi \vdash G'$ and that $T'(\rho\rho'^{-1})^{-1} = T'\rho'\rho^{-1} = T\rho^{-1} = [\![\Phi \vdash G']\!]$.
This proves the thesis. □