Mapping Fusion and Synchronized Hyperedge Replacement into Logic Programming∗

IVAN LANESE and UGO MONTANARI
Dipartimento di Informatica, Università di Pisa
Largo Bruno Pontecorvo,3 56127 Pisa, Italia
(*e-mail:* {lanese,ugo}@di.unipi.it)

submitted 27 December 2003; revised 14 April 2005; accepted 5 January 2006

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 transactional 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

∗ Work supported in part by the European IST-FET Global Computing project IST-2001-33100 PROFUNDIS and the European IST-FET Global Computing 2 project IST-2005-16004 SENSORIA.

as the standard one for this kind of systems, but there are a lot of approaches with different merits and drawbacks.

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 to study 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 attached to any number of nodes) 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). This approach has the advantage, w.r.t. other graphical frameworks such as Double Pushout (Ehrig et al. 1973) or Bigraphs (Jensen and Milner 2003), of allowing a distributed implementation since productions have a local effect and synchronization can be performed using a distributed algorithm.
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* (Lloyd 1993). 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* since many actions can be performed at the same time on different nodes 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 correspondence 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, and this allows to build chains of synchronizations. This additional power is needed to model Milner synchronization, which requires synchronous, atomic routing capabilities. 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 constrain it in order to have a close correspondence with HSHR. We define to this end *Synchronized Logic Programming* (SLP), which is a transactional version of logic programming. The idea is that function symbols are pending constraints that must be satisfied before a transaction can commit, as for zero tokens in zero-safe nets (Bruni and Montanari 2000). In the mapping from HSHR to SLP 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 (2004a) where Fusion LTS was considered instead of Fusion reduction semantics. The paper Lanese and Montanari (2002) also contains a mapping of Ambient calculus into HSHR. This result can be combined with the one here, thus obtaining a mapping of Ambient calculus into SLP. An extensive treatment of all the topics in this paper can also be found in the forthcoming Ph.D. thesis of the first author (Lanese 2006).

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 particular, since implementations of logic programming are not distributed, this can be useful mainly for simulation purposes.

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. Proofs can be found in (Lanese and Montanari 2005).

2 Background

Mathematical notation. We use $T\sigma$ to denote the application of substitution σ to T (where T can be a term or a set/vector of terms). We write substitutions as sets of pairs of the form t/x, denoting that variable x is replaced by term t. We also denote with $\sigma_1\sigma_2$ the composition of substitutions σ_1 and σ_2. We denote with $\sigma^{-1}(x)$ the set of elements mapped to x by σ. We use $|-|$ to denote the operation that computes the number of elements in a set/vector. Given a function f we denote with $\mathrm{dom}(f)$ its domain, with $\mathrm{Im}(f)$ its image and with $f|_S$ the restriction of f to the new domain S. We use on functions and substitutions set theoretic operations (such as $\cup$) referring to their representation as sets of pairs. Similarly, we apply them to vectors, referring to the set of the elements in the vector. In particular, $\setminus$ is set difference. Given a set S we denote with S^* the set of strings on S. Also, given a vector $\vec{v}$ and an integer i, $\vec{v}[i]$ is the i-th element of $\vec{v}$. Finally, a vector is given by listing its elements inside angle brackets $\langle - \rangle$.

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 Cal-

culus. 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. All these restrictions are quite standard, apart from the one concerning the match operator, which is needed to have an expansion lemma. To extend our approach to deal with match we would need to extend SHR by allowing production applications to be tagged with a unique identifier. We leave this extension for future work. 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 $u, v, \ldots, z$ and an infinite set of agent variables (disjoint w.r.t. the set of names) with meta-variable X. Names represent communication channels. We use ϕ to denote an equivalence relation on $\mathcal{N}$, called fusion, 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 w.r.t. 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). We use recursion to define infinite processes instead of other operators (e.g. replication) since it simplifies the mapping and since their expressive power is essentially the same. We use infix $+$ for binary sum (which thus is associative and commutative).

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 composition (associativity, commutativity and 0 as identity), the scope laws $(x)0 \equiv 0$, $(x)(y)P \equiv (y)(x)P$, the scope extrusion 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 $x\sigma = y\sigma$ 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 $\{\vec{x} = \vec{y}\}$ such that $\mathrm{dom}(\sigma) \subseteq \vec{z}$.

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

where σ is a substitutive effect of ϕ such that $\mathrm{dom}(\sigma) \subseteq \vec{z}$.

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

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 (hyper)edge 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 edges connected to a node expose the same action on it. For a general definition of synchronization models see Lanese and Montanari (2004b).
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

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

are matched during synchronization are unified.

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

An edge is an atomic item with a label and with as many ordered tentacles as the rank rank(L) of its label L. A set of nodes, together with a set of such edges, forms 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 the x_i are arbitrary names and rank(L) = n. Also, nil represents the empty graph and $|$ is the parallel composition of graphs (merging nodes with the same name).

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
   $G ::= L(\vec{x}) \mid G_1|G_2 \mid nil$
   where $\vec{x}$ is a vector of names and L is an edge label with rank(L) = $|\vec{x}|$.

We denote with n the function that given a graph G returns the set 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 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:

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

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

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

The well-formed judgements $\Gamma \vdash G$ over LE and $\mathcal{N}$ are those where n(G) $\subseteq \Gamma$.

Axioms (AG1),(AG2) and (AG3) define respectively the associativity, commutativity and identity over nil for operation $|$.

Well-formed judgements up to structural axioms are isomorphic to graphs up to isomorphisms. For a formal statement of the correspondence see Hirsch (2003).

We will now present the steps of a SHR computation.

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

Definition 8 (SHR transition)
Let Act be a set of actions. For each action $a \in Act$, let $\mathrm{ar}(a)$ be its arity.
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 arity 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;
- $\mathrm{n}(\pi) = \{x | \exists x' \neq x . x\pi = x'\pi\}$
  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 $\forall x \in \mathrm{n}(\Lambda).x\pi = x$, i.e. only references to representatives can be exposed. Furthermore we require $\Phi \supseteq \Gamma\pi \cup \Gamma_\Lambda$, namely nodes are never erased. Nodes in $\Gamma_{Int} = \Phi \setminus (\Gamma\pi \cup \Gamma_\Lambda)$ are fresh internal nodes, silently created in the transition. We require that no isolate, 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' thus we will have no need to write its definition explicitly in the inference rules. Notice also that we can write a SHR transition as:

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

We usually assume to have an action $\epsilon \in Act$ 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 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 behaviour of single edges.

Definition 9 (Production)
A production is a 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 considered as schemas and so they are α-convertible w.r.t. 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*)

$$\text{(par)}\quad \frac{\Gamma \vdash G_1 \xrightarrow{\Lambda,\pi} \Phi \vdash G_2 \quad \Gamma' \vdash G_1' \xrightarrow{\Lambda',\pi'} \Phi' \vdash G_2' \quad (\Gamma \cup \Phi) \cap (\Gamma' \cup \Phi') = \emptyset}{\Gamma, \Gamma' \vdash G_1 | G_1' \xrightarrow{\Lambda \cup \Lambda', \pi \cup \pi'} \Phi, \Phi' \vdash G_2 | G_2'}$$

$$\text{(merge)}\quad \frac{\Gamma \vdash G_1 \xrightarrow{\Lambda,\pi} \Phi \vdash G_2 \quad \forall x, y \in \Gamma . x\sigma = y\sigma \wedge x \neq y \Rightarrow \mathrm{act}_\Lambda(x) = \mathrm{act}_\Lambda(y)}{\Gamma\sigma \vdash G_1\sigma \xrightarrow{\Lambda',\pi'} \Phi' \vdash G_2\sigma\rho}$$

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

(i). $\rho = \mathrm{mgu}(\{(\mathrm{n}_\Lambda(x))\sigma = (\mathrm{n}_\Lambda(y))\sigma | x\sigma = y\sigma\} \cup \{x\sigma = y\sigma | x\pi = y\pi\})$ where ρ maps names to representatives in $\Gamma\sigma$ whenever possible
(ii). $\forall z \in \Gamma . \Lambda'(z\sigma) = (\Lambda(z))\sigma\rho$
(iii). $\pi' = \rho|_{\Gamma\sigma}$

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

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

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 have disjoint sets of nodes and rule (merge) allows to merge nodes (note that σ is a projection into representatives of equivalence classes). The side condition requires that we have the same action on merged nodes. Definition (i) 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 (ii) Λ is updated with these merges and that (iii) 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 (with fresh names) 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 computation.

Fig. 1. Star graph

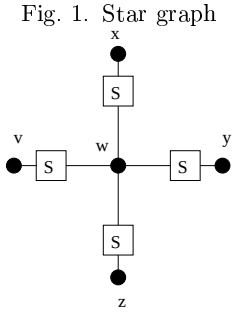
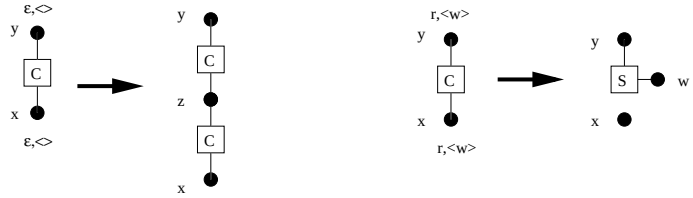


Fig. 2. Productions



Example 1 (*Hirsch et al. 2000*)
We show now 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:

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

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

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:

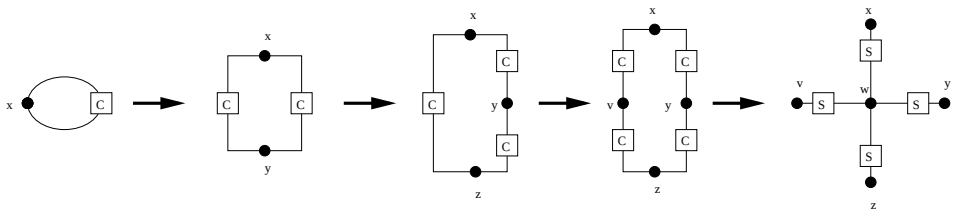
$$\begin{aligned} x \vdash C(x, x) \to x, y \vdash C(x, y)|C(y, x) \to \\ \to x, y, z \vdash C(x, y)|C(y, z)|C(z, x) \to \\ \to x, y, z, v \vdash C(x, y)|C(y, z)|C(z, v)|C(v, x) \end{aligned}$$

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

$$\begin{aligned} x, y, z, v \vdash C(x, y)|C(y, z)|C(z, v)|C(v, x) \xrightarrow{(x,r,\langle w\rangle),(y,r,\langle w\rangle),(z,r,\langle w\rangle),(v,r,\langle w\rangle)} \\ \to x, y, z, v, w \vdash S(x, w)|S(y, w)|S(z, w)|S(v, w) \end{aligned}$$

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.

Fig. 3. Ring creation and reconfiguration to star



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:
$\mathcal{P} \Vdash (\Gamma\sigma \vdash G\sigma \xrightarrow{\Lambda',\pi'} \Phi\sigma \vdash G'\sigma)$
where $\Lambda'(x\sigma) = (\Lambda(x))\sigma$ and $x\sigma\pi' = x\pi\sigma$.

Proof
By rule induction.

2.3 Logic programming

In this paper we are not interested in logic computations as refutations of goals for problem solving or artificial intelligence, but we consider logic programming (Lloyd 1993) 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 logic 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. Derivations in logic programming are called SLD-derivations (from "Linear resolution for Definite clauses with Selection function"). We will also consider partial SLD-derivations.

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

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

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

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

We will omit $P \Vdash$ if P is clear from the context.
A partial SLD-derivation of $P \cup \{G\}$ is a sequence (possibly 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 call amoeboids which implement Milner synchronization using Hoare connectors. Since Hoare synchronization involves all the edges attached to a node while Milner one involves just pairs of connectors, we use amoeboids to force each node to be shared by exactly two edges (one if the node is an interface to the outside) since in that case the behaviour of Hoare and Milner synchronization is similar. An amoeboid is essentially a router (with no path-selection abilities) 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 modelled 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 names an unbound number of times. To make the number of labels (and also of productions, as we will 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 $\rho_{S_1} = \rho\rho_{S_2}$.

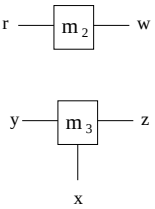
We can now define the translation from Fusion Calculus to HSHR. The translation is parametrized by the nodes in the vectors $\vec{v}$ and $\vec{w}$ we choose to represent the names in $\vec{x}$ and $\mathrm{fnarray}(P')$. We denote with $\prod_{x \in S} G_x$ the parallel composition of graphs G_x for each $x \in S$.

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

$[\![0]\!] = nil$
$[\![S]\!] = L_{\hat{S}}(x_1, \ldots, x_n)\rho_S$ with $n = |\,\mathrm{fn}(\hat{S})|$
$[\![P_1|P_2]\!] = [\![P_1]\!]|[\![P_2]\!]$
$[\![\sigma]\!] = \prod_{x \in \mathrm{Im}(\sigma)} m_{k+1}(x, \sigma^{-1}(x))$ where $k = |\sigma^{-1}(x)|$.

In the above translation, graph $[\![P']\!]$ consists of a set of disconnected edges, one for each sequential process of $(\vec{x})P'\sigma$. 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

Fig. 4. Amoeboids for σ



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 choice of the order within $\sigma^{-1}(x)$ is immaterial, since we will see that amoeboids are commutative w.r.t. 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:
$P' = \overline{u_1}z_1.0|u_2y_1.(\overline{u_3}y_2.0|\operatorname{rec} X.(x)u_4x.(\overline{u_5}x.0|X))$
$\sigma = \{u/u_1, z/z_1, u/u_2, y/y_1, u/u_3, y/y_2, u/u_4, u/u_5\}$.
We can now perform the translation.
We choose $\vec{v} = (u, z, y)$ and $\vec{w} = (u_1, z_1, u_2, y_1, u_3, y_2, u_4, u_5)$:

$$[\![(\vec{x})P'\sigma]\!]_{\vec{v},\vec{w}} = u, z, y, u_1, z_1, u_2, y_1, u_3, y_2, u_4, u_5 \vdash$$
$$L_{\overline{x_1}x_2.0}(u_1, z_1)|L_{x_1x_2.(\overline{x_3}x_4.0|\operatorname{rec} X.(x)x_5x.(\overline{x_6}x.0|X))}(u_2, y_1, u_3, y_2, u_4, u_5)|$$
$$m_6(u, u_1, u_2, u_3, u_4, u_5)|m_2(z, z_1)|m_3(y, y_1, y_2)|n(u)|n(z)|n(y)$$

Now we define the 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 transition 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 $\alpha = \phi$ 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 (which has $\{x_1, \ldots, x_n\}$ as free names). Let $\sum_i \alpha_i.P_i$ be such a process. Its productions can be derived with the following inference rule:

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

if $\vec{v} \cup \vec{w}$, $\{x_1, \ldots, x_n\}$ and $\mathrm{fn}(P_j)\xi$ are pairwise disjoint with ξ injective renaming from $\mathrm{fn}(P_j)$ to fresh names, $\Gamma' = x_1, \ldots, x_n$ and $\Gamma'' = \Gamma' \setminus (\mathrm{n}(\Lambda) \cup \mathrm{n}(\pi) \cup \mathrm{fn}(P_j))$.

We add some explanations on the derivable productions. Essentially, if $\alpha_j.P_j$ is a possible choice, the edge labelled by the process can have a transition labelled by $[\![\alpha_j]\!]$ to something related to $[\![P_j]\!]_{\vec{v},\vec{w}}$. We use $P_j\xi$ instead of P_j (and then we add the translation of ξ) to preserve the parity of the number of amoeboid edges on each path (see Definition 17). The parameter $\vec{v}$ of the translation contains fresh nodes for restricted names that are taken to the top level during the normalization of P_j while $\vec{w}$ contains the free names in the normalization of P_j (note that some of them may be duplicated w.r.t. P_j, if this one contains recursion). If α_j is a fusion ϕ, according to the semantics of the calculus, a substitutive effect π of it should be applied to P_j, and this is obtained by adding the amoeboids $[\![\pi]\!]$ in parallel. Furthermore, $\Gamma \vdash G$ must be enriched in other two ways: since nodes can never be erased, nodes which are present in the sequential process, i.e. the nodes in Γ', must be added to Γ. Also "close" n edges must be associated to forgotten nodes (to forbid further transitions on them and to have them connected to exactly two edges in the result of the transition), provided they are not exposed, i.e. to nodes in Γ''.

Note that when translating the RHS $(\vec{x})P\sigma$ of productions we may have names in $P\sigma$ which occur just once. Since they are renamed by σ and ξ, they will produce in the translation some chains of m_2 connectors of even length, which, as we will see shortly, are behaviourally equivalent to simple nodes. For simplicity, in the examples we will use the equivalent productions where these connectors have been removed and the nodes connected by them have been 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 (where $\vec{v} = \vec{w} = \langle\rangle$) is:
$x_1, x_2 \vdash L_{\overline{x_1}x_2.0}(x_1, x_2) \xrightarrow{(x_1, out_1, \langle x_2 \rangle)} x_1, x_2 \vdash n(x_1)$
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:
$x_1x_2.(\overline{x_3}x_4.0|\,\mathrm{rec}\, X.(x)x_5x.(\overline{x_6}x.0|X))$.
The process $\overline{x_3}x_4.0|\,\mathrm{rec}\, X.(x)x_5x.(\overline{x_6}x.0|X)\xi$ where $x_i\xi = x'_i$ can be transformed

into:
$(y)\overline{y_1}y_2.0|y_3y_4.(\overline{y_5}y_6.0|\operatorname{rec} X.((x)y_7x.(\overline{y_8}x.0|X)))\sigma$
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 (with $\vec{v} = \langle y \rangle$ and $\vec{w} = \langle y_1, y_2, y_3, y_4, y_5, y_6, y_7, y_8 \rangle$) is:

$$x'_3, x'_4, x'_5, x'_6, y_1, y_2, y_3, y_4, y_5, y_6, y_7, y_8, y \vdash$$
$$L_{\overline{x_1}x_2.0}(y_1, y_2)|L_{x_1x_2.(\overline{x_3}x_4.0|\operatorname{rec} X.((x)x_5x.(\overline{x_6}x.0|X)))}(y_3, y_4, y_5, y_6, y_7, y_8)|$$
$$m_2(x'_3, y_1)|m_2(x'_4, y_2)|m_3(x'_5, y_3, y_7)|m_3(x'_6, y_5, y_8)|m_3(y, y_4, y_6)|n(y)$$

Thus the production is:

$$x_1, x_2, x_3, x_4, x_5, x_6 \vdash L_{x_1x_2.(\overline{x_3}x_4.0|\operatorname{rec} X.(x)x_5x.(\overline{x_6}x.0|X))}(x_1, x_2, x_3, x_4, x_5, x_6)$$
$$\xrightarrow{(x_1, in_1, \langle x_2 \rangle)}$$
$$x_1, x_2, x_3, x_4, x_5, x_6, y_3, y_4, y_5, y_6, y_7, y_8, y, x'_5, x'_6 \vdash$$
$$L_{\overline{x_1}x_2.0}(x_3, x_4)|L_{x_1x_2.(\overline{x_3}x_4.0|\operatorname{rec} X.((x)x_5x.(\overline{x_6}x.0|X)))}(y_3, y_4, y_5, y_6, y_7, y_8)|$$
$$m_3(x'_5, y_3, y_7)|m_3(x'_6, y_5, y_8)|m_3(y, y_4, y_6)|n(y)|m_2(x_5, x'_5)|m_2(x_6, x'_6)|n(x_1)$$

where for simplicity we collapsed y_1 with x_3 and y_2 with x_4.

We will now show the productions for amoeboids.

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

$$\Gamma \vdash m_k(\Gamma) \xrightarrow{(x_1, in_n, \vec{y}_1), (x_2, out_n, \vec{y}_2)} \Gamma, \vec{y}_1, \vec{y}_2 \vdash m_k(\Gamma)|\prod_{i=1\ldots|\vec{y}_1|} m_2(\vec{y}_1[i], \vec{y}_2[i])$$

We need such a production for each k and n and each pair of nodes x_1 and x_2 in Γ where Γ is a chosen tuple of distinct names with k components and $\vec{y}_1$ 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 w.r.t. permutations of the tentacles, modelling the fact that its tentacles are essentially unordered.
We have no productions for edges labelled with n, which thus forbid any synchronization.

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 present a more general definition of amoeboid for a set of nodes and we show that, in the situations of interest, these amoeboids behave exactly as the simpler m_i edges.

Definition 17 (*Structured amoeboid*)
Given a vector of nodes $\vec{s}$, a structured amoeboid $M(\vec{s})$ for the set of nodes S containing all the nodes in $\vec{s}$ is any connected graph composed by m and n edges that satisfies the following properties:

- its set of nodes is of the form $S \cup I$, with $S \cap I = \emptyset$;
- nodes in S are connected to exactly one edge of the amoeboid;

- nodes in I are connected to exactly two edges of the amoeboid;
- the number of edges composing each path connecting two distinct nodes of S is odd.

Nodes in S are called *external*, nodes in I are called *internal*. We consider *equivalent* all the amoeboids with the same set S of external nodes. The last condition is required since each connector inverts the polarity of the synchronization, and we want amoeboids to invert it.
Note that $m_{|S|}(\vec{s})$ is an amoeboid for S.

Lemma 2
If $M(\vec{s})$ is a structured amoeboid for S, the transitions for $M(\vec{s})$ which are non idle and expose non ϵ actions on at most two nodes $x_1, x_2 \in S$ are of the form:

$$S, I \vdash M(\vec{s}) \xrightarrow{\Lambda, id} S, I, I', \vec{y}_1, \vec{y}_2 \vdash M(\vec{s}) | \prod_{i=1\ldots|\vec{y}_1|} M(\vec{y}_1[i], \vec{y}_2[i]) | \prod \tilde{M}(\emptyset)$$

where $\Lambda(x_1) = (in_n, \vec{y}_1)$ and $\Lambda(x_2) = (out_n, \vec{y}_2)$ (non trivial actions may be exposed also on some internal nodes) and $\vec{y}_1$ and $\vec{y}_2$ are two vectors of fresh names such that $|\vec{y}_1| = |\vec{y}_2| = n$. Here $\tilde{M}(\emptyset)$ contains rings of m_2 connectors connected only to fresh nodes which thus are disconnected from the rest of the graph. We call them *pseudoamoeboids*. Furthermore we have at least one transition of this kind for each choice of x_1, x_2, $\vec{y}_1$ and $\vec{y}_2$.

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 1 (*Correctness*)
For each closed fusion process P and each pair of vectors $\vec{v}$ and $\vec{w}$ satisfying the constraints of Definition 13, if $P \rightarrow P'$ 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 $[\![P']\!]_{\vec{v'},\vec{w'}}$ (for some $\vec{v'}$ and $\vec{w'}$) up to isolated nodes, up to injective renamings, up to equivalence of amoeboids ($\Gamma \vdash G$ can have a structured amoeboid where $[\![P']\!]_{\vec{v'},\vec{w'}}$ has a simple one) and up to pseudoamoeboids.

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

Theorem 2 (*Completeness*)
For each closed fusion process P and each pair of vectors $\vec{v}$ and $\vec{w}$ if $[\![P]\!]_{\vec{v},\vec{w}} \xrightarrow{\Lambda, \pi} \Gamma \vdash G$ with a 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 $[\![P']\!]_{\vec{v'},\vec{w'}}$ (for some $\vec{v'}$ and $\vec{w'}$) up to isolate nodes, up to injective renamings, up to equivalence of amoeboids ($\Gamma \vdash G$ can have a structured amoeboid where $[\![P']\!]_{\vec{v'},\vec{w'}}$ has a simple one) and up to pseudoamoeboids.

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. On the contrary, one can consider the fusion equivalent of all the HSHR transitions: these correspond to concurrent executions of many fusion reductions. One can give a semantics for Fusion Calculus with that behaviour. Anyway in that case the notion of equivalence of amoeboids is no more valid, since different amoeboids allow different degrees of concurrency. We thus need to constrain them. The simplest case is to have only simple amoeboids, that is to have no concurrency inside a single channel, but there is no way to force normalization of amoeboids to happen before undesired transitions can occur. The opposite case (all the processes can interact in pairs, also on the same channel) can be realized, but it requires more complex auxiliary productions.

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; isolated nodes and pseudoamoeboids are not relevant since different connected components evolve independently). Thus the previous results can be extended from transitions to whole computations.

Note that in the HSHR model the behavioural part of the system is represented by productions while the topological part is represented by graphs. 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:

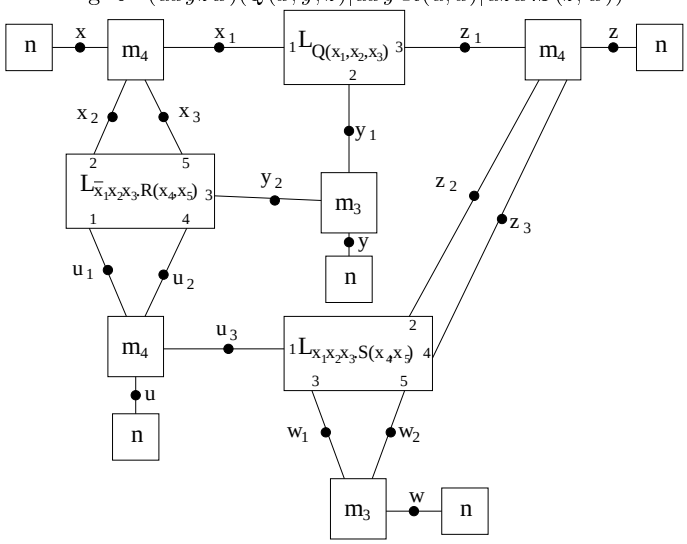
$$(uxyzw)(Q(x,y,z)|\overline{u}xy.R(u,x)|uzw.S(z,w))$$

Note that it is already in the form $(\vec{x})P$. It can do the following transition:

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

We can write P in the form:
$(Q(x_1,y_1,z_1)|\overline{u_1}x_2y_2.R(u_2,x_3)|u_3z_2w_1.S(z_3,w_2))\sigma$
where:
$\sigma = \{x/x_1, y/y_1, z/z_1, u/u_1, x/x_2, y/y_2, u/u_2, x/x_3, u/u_3, z/z_2, w/w_1, z/z_3, w/w_2\}$.

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



A translation of the starting process is:

$$u,x,y,w,z,x_1,y_1,z_1,u_1,x_2,y_2,u_2,x_3,u_3,z_2,w_1,z_3,w_2 \vdash$$
$$L_{Q(x_1,x_2,x_3)}(x_1,y_1,z_1)|L_{\overline{x_1}x_2x_3.R(x_4,x_5)}(u_1,x_2,y_2,u_2,x_3)|$$
$$L_{x_1x_2x_3.S(x_4,x_5)}(u_3,z_2,w_1,z_3,w_2)|m_4(u,u_1,u_2,u_3)|m_4(x,x_1,x_2,x_3)|$$
$$m_3(y,y_1,y_2)|m_4(z,z_1,z_2,z_3)|m_3(w,w_1,w_2)|n(u)|n(x)|n(y)|n(w)|n(z)$$

A graphical representation is in Figure 5.

We have the following process productions:

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

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

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 u_1 and u_3 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:

$$u,x,y,w,z,x_1,y_1,z_1,u_1,x_2,y_2,u_2,x_3,u_3,z_2,w_1,z_3,w_2 \vdash$$
$$L_{Q(x_1,x_2,x_3)}(x_1,y_1,z_1)|L_{R(x_1,x_2)}(u_2,x_3)|n(u_1)|L_{S(x_1,x_2)}(z_3,w_2)|n(u_3)|$$
$$m_4(u,u_1,u_2,u_3)|m_4(x,x_1,x_2,x_3)|m_3(y,y_1,y_2)|m_4(z,z_1,z_2,z_3)|m_3(w,w_1,w_2)|$$
$$m_2(x_2,z_2)|m_2(y_2,w_1)|n(u)|n(x)|n(y)|n(w)|n(z)$$

Fig. 6. Graph with actions

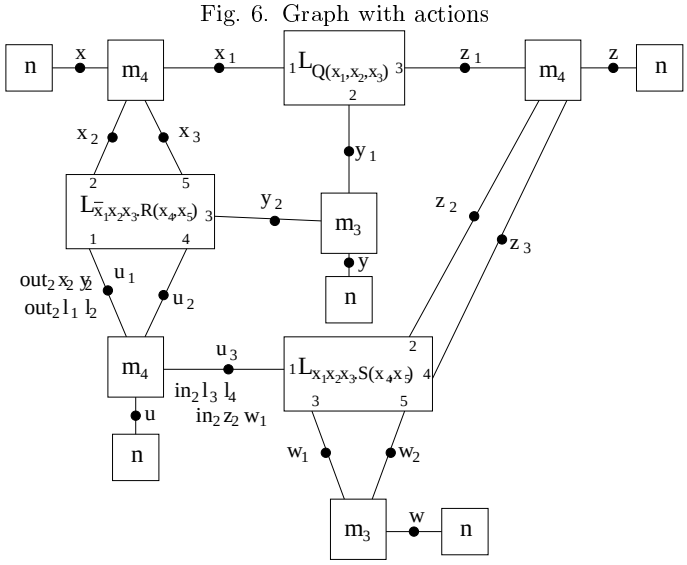
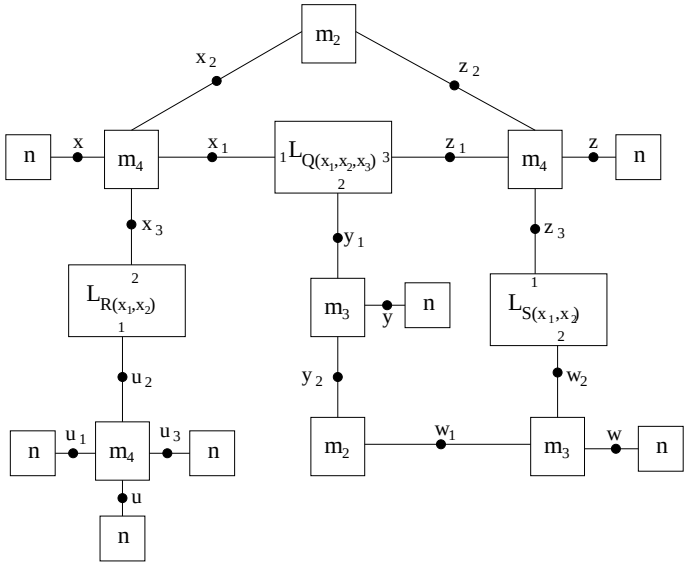


Fig. 7. Resulting graph



which is represented in Figure 7.

The amoeboids connect the following tuples of nodes:
(u, u_1, u_2, u_3), $(x, x_1, x_2, x_3, z_2, z, z_1, z_3)$, $(w, w_1, w_2, y_2, y, y_1)$. 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 $(uxy)Q(x, y, x)|R(u, x)|S(x, y)$ 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 $(uz)\overline{u}z|\operatorname{rec} X.(x)ux.(\overline{u}x.0|X)$. The translation of this process, as shown in Example

3 is:

$$u, z, y, u_1, z_1, u_2, y_1, u_3, y_2, u_4, u_5 \vdash$$
$$L_{\overline{x_1}x_2.0}(u_1, z_1) | L_{x_1x_2.(\overline{x_3}x_4.0 | \, \mathrm{rec}\, X.(x)x_5x.(\overline{x_6}x.0 | X))}(u_2, y_1, u_3, y_2, u_4, u_5) |$$
$$m_6(u, u_1, u_2, u_3, u_4, u_5) | m_2(z, z_1) | m_3(y, y_1, y_2) | n(u) | n(z) | n(y)$$

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

$$u_1, z_1 \vdash L_{\overline{x_1}x_2.0}(u_1, z_1) \xrightarrow{(u_1, out_1, \langle z_1 \rangle)} u_1, z_1 \vdash n(u_1)$$

$$u_2, y_1, u_3, y_2, u_4, u_5 \vdash L_{x_1x_2.(\overline{x_3}x_4.0 | \, \mathrm{rec}\, X.(x)x_5x.(\overline{x_6}x.0 | X))}(u_2, y_1, u_3, y_2, u_4, u_5)$$
$$\xrightarrow{u_2, in_1, \langle y_1 \rangle}$$
$$u_2, y_1, u_3, y_2, u_4, u_5, w_1, w_2, w_3, w_4, w_5, w_6, y', u'_4, u'_5 \vdash$$
$$L_{\overline{x_1}x_2.0}(u_3, y_2) | L_{x_1x_2.(\overline{x_3}x_4.0 | \, \mathrm{rec}\, X.((x)x_5x.(\overline{x_6}x.0 | X)))}(w_1, w_2, w_3, w_4, w_5, w_6) |$$
$$m_3(u'_4, w_1, w_5) | m_3(u'_5, w_3, w_6) | m_3(y', w_2, w_4) | n(y') | m_2(u_4, u'_4) | m_2(u_5, u'_5) | n(u_2)$$

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

$$u, z, y, u_1, z_1, u_2, y_1, u_3, y_2, u_4, u_5 \vdash$$
$$L_{\overline{x_1}x_2.0}(u_1, z_1) | L_{x_1x_2.(\overline{x_3}x_4.0 | \, \mathrm{rec}\, X.(x)x_5x.(\overline{x_6}x.0 | X))}(u_2, y_1, u_3, y_2, u_4, u_5) |$$
$$m_6(u, u_1, u_2, u_3, u_4, u_5) | m_2(z, z_1) | m_3(y, y_1, y_2) | n(u) | n(z) | n(y)$$
$$\xrightarrow{(u_1, out_1, \langle z_1 \rangle)(u_2, in_1, \langle y_1 \rangle)}$$
$$u, y, z, x, u_1, z_1, u_2, y_1, u_3, y_2, u_4, u_5, w_1, w_2, w_3, w_4, w_5, w_6, y', u'_4, u'_5 \vdash$$
$$n(u_1) | L_{\overline{x_1}x_2.0}(u_3, y_2) | L_{x_1x_2.(\overline{x_3}x_4.0 | \, \mathrm{rec}\, X.((x)x_5x.(\overline{x_6}x.0 | X)))}(w_1, w_2, w_3, w_4, w_5, w_6) |$$
$$m_3(u'_4, w_1, w_5) | m_3(u'_5, w_3, w_6) | m_3(y', w_2, w_4) | n(y') | m_2(u_4, u'_4) | m_2(u_5, u'_5) | n(u_2) |$$
$$m_6(u, u_1, u_2, u_3, u_4, u_5) | m_2(z_1, y_1) | m_2(z, z_1) | m_3(y, y_1, y_2) | n(u) | n(z) | n(y)$$

The resulting graph is, up to injective renaming and equivalence of amoeboids, a translation of:
$(uyy')(\overline{u}y.0 | uy'.(\overline{u}y'.0 | \, \mathrm{rec}\, X.(x)ux.(\overline{u}x.0 | X)))$
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

As shown in the table, we represent (closed) processes by graphs where edges are sequential processes and amoeboids model names. The inactive process 0 is the empty graph nil. From a dynamic point of view, Fusion reductions are modelled by HSHR transitions obtained composing productions that represent prefix executions.

4 Mapping Hoare SHR into logic programming

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 Synchronized Logic Programming

In this subsection we present Synchronized Logic Programming.
SLP has been introduced because logic programming allows for many execution strategies and for complex interactions. Essentially SLP is obtained from standard logic programming by adding a mechanism of transactions. The approach is similar to the zero-safe nets approach (Bruni and Montanari 2000) for Petri nets. In particular we consider that function symbols are resources that can be used only inside a transaction. A transaction can thus end only when the goal contains just predicates and variables. During a transaction, which is called big-step in this setting, each atom can be rewritten at most once. If a transaction can not be terminated, then the computation is not allowed. A computation is thus a sequence of big-steps.

This synchronized flavour of logic programming corresponds to HSHR since:

- used goals correspond to graphs (goal-graphs);
- clauses in programs correspond to HSHR productions (synchronized clauses);
- resulting computations model HSHR computations (synchronized computations).

Definition 18 (*Goal-graph*)
We call goal-graph a goal which has no function symbols (constants are considered as functions of arity 0).

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 (of arity at least 1) 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 since $p(x, f(y))$ is not a goal-graph;
$q(g(f(x)), y) \leftarrow p(x, y)$	not synchronized since it contains nested functions;
$q(f(x), y, f(z)) \leftarrow p(x)$	not synchronized since 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, even if z does not appear in the body.

In the mapping, the transaction mechanism is used to model the synchronization of HSHR, where edges can be rewritten only if the synchronization constraints are satisfied. In particular, a clause $A(t_1, \ldots, t_n) \leftarrow B_1, \ldots, B_n$ will represent 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 (this would correspond to allow an action in a production to synchronize with a sequence of actions from a computation of an adjacent subgraph). 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.

[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 disconnect from 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. An explicit translation of action ϵ can be used too.

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 are 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. For simplicity, we suppose that the set of nodes in the SHR model coincides with the set of variables in SLP (otherwise we need a bijective translation function). We do the same for edge labels and names of predicates, and for actions and function symbols.

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 $G \equiv G'$.

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 3 (*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 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. $\Box$

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

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

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

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

The idea of the translation is that the condition given by an action $(x, a, \vec{y})$ is represented by using the term $a(x\pi, \vec{y})$ as argument in the position that corresponds to x. Notice that in this term a is a function symbol and π is a substitution. During unification, x will be bound to that term and, when other instances of x are met, the corresponding term must contain the same function 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 variable 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 variables are automatically built when the clauses are applied. 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 variables. In fact in HSHR nodes are mapped to their representatives according to π, while, in SLP, θ cannot do the same, since the variables of the clause variant must be all fresh. The possible choices of fresh names for the variables change by an injective renaming 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:
$\theta_\rho = \{(a(x\pi\rho, \vec{y}\rho)/x | \Lambda(x) = (a, \vec{y}), a \neq \epsilon\} \cup \{x\pi\rho/x | \Lambda(x) = (\epsilon, \langle\rangle)\}$
for some injective renaming ρ.

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

Theorem 4 (*Correctness*)
Let $\mathcal{P}$ be a set of productions of a 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 $x\rho$ is a fresh variable unless possibly when $x \in \Gamma \wedge \Lambda(x) = (\epsilon, \langle\rangle)$. In that case we may have $x\rho = x$. Furthermore θ_ρ is associated to $\Gamma \vdash G \xrightarrow{\Lambda,\pi} \Phi \vdash G'$ and $T = [\![\Phi \vdash G']\!]\rho$. Finally, used productions translate into the clauses used in the big-step and are applied to the edges that translate into the predicates rewritten by them.

Proof
The proof is by rule induction.

Theorem 5 (Completeness)
Let $\mathcal{P}$ be a set of productions of a 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:
$[\![\Gamma \vdash G]\!] \stackrel{\theta}{\Rightarrow} T$
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 $T = [\![\Phi \vdash G']\!]\rho$ and $\mathcal{P} \Vdash (\Gamma \vdash G \xrightarrow{\Lambda,\pi} \Phi \vdash G')$.

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

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

Remember that a translation of the starting process is:

$$u,x,y,w,z,x_1,y_1,z_1,u_1,x_2,y_2,u_2,x_3,u_3,z_2,w_1,z_3,w_2 \vdash L_{Q(x_1,x_2,x_3)}(x_1,y_1,z_1)|L_{\overline{x_1}x_2x_3.R(x_4,x_5)}(u_1,x_2,y_2,u_2,x_3)| L_{x_1x_2x_3.S(x_4,x_5)}(u_3,z_2,w_1,z_3,w_2)|m_4(u,u_1,u_2,u_3)|m_4(x,x_1,x_2,x_3)| m_3(y,y_1,y_2)|m_4(z,z_1,z_2,z_3)|m_3(w,w_1,w_2)|n(u)|n(x)|n(y)|n(w)|n(z)$$

We have the following productions:

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

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

that corresponds to the clauses (we directly write suitably renamed variants):
$L_{\overline{x_1}x_2x_3.R(x_4,x_5)}(out_2(u'_1,x'_2,y'_2),x'_2,y'_2,u'_2,x'_3) \leftarrow L_{R(x_1,x_2)}(u'_2,x'_3)|n(u'_1)$
$L_{x_1x_2x_3.S(x_4,x_5)}(in_2(u'''_3,z'''_2,w'''_1),z'''_2,w'''_1,z'''_3,w'''_2) \leftarrow L_{S(x_1,x_2)}(z'''_3,w'''_2)|n(u'''_3)$
plus the clause obtained from the auxiliary production:

$$m_4(u'',out_2(u''_1,x''_2,y''_2),u''_2,in_2(u''_3,z''_2,w''_1)) \leftarrow m_4(u'',u''_1,u''_2,u''_3),m_2(x''_2,z''_2),m_2(y''_2,w''_1)$$

We obtain the big-step represented in Figure 8. The observable substitution of the big-step is $\{out_2(u'_1,x_2,y_2)/u_1, in_2(u''_3,z_2,w_1)/u_3\}$. This is associated to the wanted HSHR transition with $\rho = \{u'_1/u_1, u''_3/u_3\}$ and by applying ρ to the final graph of the HSHR transition we obtain:

$$L_{Q(x_1,x_2,x_3)}(x_1,y_1,z_1)|L_{R(x_1,x_2)}(u_2,x_3)|n(u'_1)|L_{S(x_1,x_2)}(z_3,w_2)|n(u''_3)| m_4(u,u'_1,u_2,u''_3)|m_4(x,x_1,x_2,x_3)|m_3(y,y_1,y_2)|m_4(z,z_1,z_2,z_3)|m_3(w,w_1,w_2)| m_2(x_2,z_2)|m_2(y_2,w_1)|n(u)|n(x)|n(y)|n(w)|n(z)$$

Fig. 8. Big-step for a Fusion transition

$$L_{Q(x_1,x_2,x_3)}(x_1,y_1,z_1), L_{\overline{x_1}x_2x_3.R(x_4,x_5)}(u_1,x_2,y_2,u_2,x_3),$$
$$L_{x_1x_2x_3.S(x_4,x_5)}(u_3,z_2,w_1,z_3,w_2),$$
$$m_4(u,u_1,u_2,u_3), m_4(x,x_1,x_2,x_3), m_3(y,y_1,y_2), m_4(z,z_1,z_2,z_3), m_3(w,w_1,w_2),$$
$$n(u), n(x), n(y), n(w), n(z)$$
$$\xrightarrow{out_2(u_1',x_2,y_2)/u_1, x_2/x_2', y_2/y_2', u_2/u_2', x_3/x_3'}$$
$$L_{Q(x_1,x_2,x_3)}(x_1,y_1,z_1), L_{R(x_1,x_2)}(u_2,x_3), n(u_1'), L_{x_1x_2x_3.S(x_4,x_5)}(u_3,z_2,w_1,z_3,w_2),$$
$$m_4(u,out_2(u_1',x_2,y_2),u_2,u_3), m_4(x,x_1,x_2,x_3), m_3(y,y_1,y_2),$$
$$m_4(z,z_1,z_2,z_3), m_3(w,w_1,w_2), n(u), n(x), n(y), n(w), n(z)$$
$$\xrightarrow{u/u'', u_1'/u_1'', x_2/x_2'', y_2/y_2'', u_2/u_2'', in_2(u_3'',z_2'',w_1'')/u_3}$$
$$L_{Q(x_1,x_2,x_3)}(x_1,y_1,z_1), L_{R(x_1,x_2)}(u_2,x_3), n(u_1'),$$
$$L_{x_1x_2x_3.S(x_4,x_5)}(in_2(u_3'',z_2'',w_1''), z_2, w_1, z_3, w_2),$$
$$m_4(u,u_1',u_2,u_3''), m_2(x_2,z_2''), m_2(y_2,w_1''), m_4(x,x_1,x_2,x_3), m_3(y,y_1,y_2),$$
$$m_4(z,z_1,z_2,z_3), m_3(w,w_1,w_2), n(u), n(x), n(y), n(w), n(z)$$
$$\xrightarrow{u_3''/u_3''', z_2/z_2'', w_1/w_1'', z_2/z_2''', w_1/w_1''', z_3/z_3''', w_2/w_2'''}$$
$$L_{Q(x_1,x_2,x_3)}(x_1,y_1,z_1), L_{R(x_1,x_2)}(u_2,x_3), n(u_1'), L_{S(x_1,x_2)}(z_3,w_2), n(u_3''),$$
$$m_4(u,u_1',u_2,u_3''), m_2(x_2,z_2), m_2(y_2,w_1), m_4(x,x_1,x_2,x_3), m_3(y,y_1,y_2),$$
$$m_4(z,z_1,z_2,z_3), m_3(w,w_1,w_2), n(u), n(x), n(y), n(w), n(z)$$

that, translated, becomes the final goal of the big-step 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	$\square$
Production	Clause	Action	Function s.

Essentially the correspondence is given by the homomorphism between graphs and goals, with edges mapped to atomic goals, nodes to variables, parallel composition to And composition and *nil* to $\square$. Dynamically, HSHR transitions are modelled by big-steps, that are transactional applications of clauses which model productions. Finally, HSHR actions are modelled by function symbols.

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 w.r.t. 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 transactional version 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, which considers the LTS of Fusion Calculus instead of reductions (see Lanese and Montanari 2004a).

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 behaviour 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 (Sangiorgi 1993), 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 of 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 the paper. Furthermore our operational correspondence between HSHR and SLP is very strong and it should respect any reasonable abstract semantics. The mapping from Fusion Calculus into HSHR deals only with closed terms, thus no observations can be considered. 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 restrict 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, allowing to exploit existing ideas, algorithms and tools developed for logic programming.

References

Bruni, R. and Montanari, U. 2000. Zero-safe nets: Comparing the collective and individual token approaches. *Information and Computation 156,* 1-2, 46–89.

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, Parallelism, and Distribution.* World Scientific.

Ehrig, H., Pfender, M., and Schneider, H. J. 1973. Graph grammars: an algebraic

approach. In *Proc. of IEEE Conference on Automata and Switching Theory*. IEEE Computer Society, 167–180.

Ferrari, G. L., Montanari, U., and Tuosto, E. 2001. A LTS semantics of ambients via graph synchronization with mobility. In *Proc. of ICTCS'01*. 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 *Proc. of FoSSaCS'04*. LNCS, vol. 2987. Springer, 484–498.

Hirsch, D. 2003. Graph transformation models for software architecture styles. Ph.D. thesis, Departamento de Computación, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires, Argentina.

Hirsch, D., Inverardi, P., and Montanari, U. 2000. Reconfiguration of software architecture styles with name mobility. In *Proc. of COORDINATION 2000*. LNCS, vol. 1906. Springer, 148–163.

Hirsch, D. and Montanari, U. 2001. Synchronized hyperedge replacement with name mobility. In *Proc. of CONCUR'01*. LNCS, vol. 2154. Springer, 121–136.

Jensen, O. H. and Milner, R. 2003. Bigraphs and transitions. *SIGPLAN Not. 38,* 1, 38–49.

König, B. and Montanari, U. 2001. Observational equivalence for synchronized graph rewriting. In *Proc. of TACS'01*. LNCS, vol. 2215. Springer, 145–164.

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

Lanese, I. 2006. Synchronization strategies for global computing models. Ph.D. thesis, Computer Science Department, University of Pisa, Pisa, Italy. Forthcoming.

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*. Anais, 11–35.

Lanese, I. and Montanari, U. 2004a. A graphical fusion calculus. In *Proc. of the Workshop of the COMETA Project on Computational Metamodels*. Electronic Notes in Theoretical Computer Science, vol. 104. Elsevier Science, 199–215.

Lanese, I. and Montanari, U. 2004b. Synchronization algebras with mobility for graph transformations. In *Proc. of FGUC'04 – Foundations of Global Ubiquitous Computing*. Electronic Notes in Theoretical Computer Science, vol. 138. Elsevier Science, 43–60.

Lanese, I. and Montanari, U. 2005. Mapping fusion and synchronized hyperedge replacement into logic programming. Computing Research Repository (`http://arxiv.org/corr/home`). Paper n. cs.LO/0504050.

Lloyd, J. W. 1993. *Foundations of Logic Programming, Second Extended Edition*. Springer.

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

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

Sangiorgi, D. 1993. A theory of bisimulation for the pi-calculus. In *Proc. of CONCUR'93*. LNCS, vol. 715. Springer, 127–142.

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