Insights emerged while comparing three models for global computing

Ivan Lanese and Ugo Montanari*

University of Pisa, Computer Science Department
Largo B. Pontecorvo,3 56127 Pisa Italy
{lanese,ugo}@di.unipi.it

Abstract. In this paper we outline the main ideas emerged while studying a chain of mappings from *Fusion Calculus* to *logic programming*, using *Synchronized Hyperedge Replacement* (with both Hoare and Milner synchronizations) as intermediate step. We aim more at discussing the ideas behind the mappings than at presenting their technical details.

Keywords. Fusion Calculus, graph transformation, Synchronized Hyperedge Replacement, logic programming, synchronization, mobility.

1 Introduction

These years have seen many efforts toward the goal of finding a formal model able to deal with all the challenges posed by global computing (GC) systems.

We recall here some aspects of GC systems we are particularly interested in:

- size of the system: GC systems are usually composed by a large number of different components, thus special care is required in order to make their models tractable;
- distribution: locations are an important aspect of system structure, which must be modeled explicitly;
- mobility: hardware and software entities can change their location at runtime, thus mobility primitives are required in the model;
- open-endness: usually models deal with partial systems, which are supposed to interact with an environment, thus models of different systems must be composable;
- reconfigurability: changes in the structure of the system are one of the main effects of the behaviour of GC components, which must be reflected by models.

Thus we want to find models for the coordination (computation is usually abstracted away) of different interacting entities, deployed on huge scales and with evolving structure.

Until now, no model for GC systems has emerged as the best one, thus we think it is useful to compare the different existing ones, in order to understand

* I. Lanese has been supported by EU-FET project **AGILE** IST-2001-32747. U. Montanari has been supported by EU-FET project **PROFUNDIS** IST-2001-33100.

Dagstuhl Seminar Proceedings 05081
Foundations of Global Computing
http://drops.dagstuhl.de/opus/volltexte/2006/295

their relationships, their strengths and their weaknesses and find useful insights on how to build new ones. Since a comparison of all the proposals in the literature is far beyond our possibilities, we concentrate on three of them:

- Fusion Calculus [1,2]: it is a process calculus similar to the π-calculus [3], with the additional ability of merging names;
- Synchronized Hyperedge Replacement (SHR) [4,5]: it is a graph transformation framework, where transitions are obtained by synchronizing productions that describe how single hyperedges evolve. Notably, SHR productions can be synchronized using different synchronization models;
- logic programming [6]: while widely known as problem-solving language, logic programming is also an interesting goal-rewriting engine, and it is used here with this aim.

Structure of the paper. Section 2 introduces the three formalisms to be compared, namely Fusion Calculus (2.1), Synchronized Hyperedge Replacement (2.2) and logic programming (2.3). Section 3 presents the mappings: from Fusion Calculus to Milner SHR (3.1), from Milner SHR to Hoare SHR (3.2) and finally from Hoare SHR to logic programming (3.3). Conclusions and traces for future work are collected in Section 4. A more technical but probably less insightful presentation of a large part of this work (but without Milner SHR and restriction for hypergraphs) can be found in [7].

2 The three formalisms

We present here the main features of the three formalisms, while referring to the literature for a detailed presentation of them.

2.1 Fusion Calculus

We consider the calculus as presented in [1].

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 $u, v, \ldots, z$ 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*. 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 α are defined by:*

$$\begin{array}{rll} \alpha ::= & u\boldsymbol{x} & \textit{(Input)} \\ & \overline{u}\boldsymbol{x} & \textit{(Output)} \\ & \phi & \textit{(Fusion)} \end{array}$$

where $\boldsymbol{x}$ is a tuple of names.

Definition 2. *The sequential agents S and the general agents P are defined by:*

$$S ::= \sum_i \alpha_i.P_i \text{ (Guarded sum)} \qquad P ::= \begin{array}{ll} 0 & \text{(Inaction)} \\ S & \text{(Sequential agent)} \\ P_1|P_2 & \text{(Composition)} \\ (x)P & \text{(Scope)} \\ \mathrm{rec}\, X.P & \text{(Recursion)} \\ X & \text{(Agent variable)} \end{array}$$

The scope restriction operator is the only binder for names. Similarly rec is the binder for agent variables. We will only consider agents which are closed w.r.t. agent variables and with guarded-recursion. 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. A process is sequential iff in the equivalence class it stands for there is at least a sequential agent.

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 $\sigma : \mathcal{N} \to \mathcal{N}$ such that $x\sigma = y\sigma$ iff $x\phi y$.*

Note that σ sends all members of each equivalence class of ϕ to one representative in the class, and it is a most general unifier of ϕ, when this is considered as a set of equations.

Definition 5. *A bound action is of the form $(\boldsymbol{z})a\boldsymbol{x}$ where $|\boldsymbol{z}| > 0$ and all elements in $\boldsymbol{z}$ are also in $\boldsymbol{x}$. Names in $\boldsymbol{z}$ are bound names. The actions consist of the free actions and the bound actions.*

For convenience we define $\phi \setminus z$ to mean $\phi \cap (\mathcal{N} \setminus \{z\})^2 \cup \{(z, z)\}$.

Definition 6 (SOS semantics for Fusion Calculus). *The SOS semantics for Fusion Calculus is the least labelled transition system generated by the inference rules in Table 1.*

$$\text{(PREF)}\ \alpha.P \xrightarrow{\alpha} P \qquad \text{(SUM)}\ \frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'}$$

$$\text{(PAR)}\ \frac{P \xrightarrow{\alpha} P'}{P|Q \xrightarrow{\alpha} P'|Q} \qquad \text{(COM)}\ \frac{P \xrightarrow{u\boldsymbol{x}} P' \quad Q \xrightarrow{\overline{u}\boldsymbol{y}} Q' \quad |\boldsymbol{x}| = |\boldsymbol{y}|}{P|Q \xrightarrow{\{\boldsymbol{x}=\boldsymbol{y}\}} P'|Q'}$$

$$\text{(SCOPE)}\ \frac{P \xrightarrow{\phi} P' \quad z\phi x \quad z \neq x}{\nu z\ P \xrightarrow{\phi\setminus z} P'\{x/z\}} \qquad \text{(OPEN)}\ \frac{P \xrightarrow{(\boldsymbol{y})a\boldsymbol{x}} P' \quad z \in \boldsymbol{x} \setminus \boldsymbol{y} \quad a \notin \{z, \overline{z}\}}{\nu z\ P \xrightarrow{(z\boldsymbol{y})a\boldsymbol{x}} P'}$$

$$\text{(PASS)}\ \frac{P \xrightarrow{\alpha} P' \quad z \notin \mathrm{n}(\alpha)}{\nu z\ P \xrightarrow{\alpha} \nu z\ P'} \qquad \text{(STRUCT)}\ \frac{P \xrightarrow{\alpha} P' \quad P \equiv Q \quad P' \equiv Q'}{Q \xrightarrow{\alpha} Q'}$$

Table 1. Inference rules for Fusion Calculus.

2.2 Synchronized Hyperedge Replacement

Synchronized Hyperedge Replacement (SHR) [4,5] is a framework for *hypergraph transformations* developed in order to be able to define complex reconfigurations using *productions* with a local effect (which can thus be easily implemented also in a distributed setting) and a *synchronization mechanism* with also built-in node *mobility*.

Definition 7 (Hypergraph). *A (hyper)graph is composed by a set of labelled (hyper)edges, a set of nodes and a connection function that associates to each edge a tuple of nodes whose cardinality is determined by the rank* $\mathrm{rank}(L)$ *of its label* L*. These are called the attachment nodes of the edge. We consider graphs with an interface composed by a subset of the nodes, identified by their names.*

We use for graphs a textual representation, that makes easier to define transitions.

Definition 8 (Textual representation for graphs). *Let* $\mathcal{N}$ *be a fixed infinite set of names and* LE *a ranked alphabet of labels. We represent a graph as* $\Gamma \vdash G$ *where:*

1. $\Gamma \subseteq \mathcal{N}$ *is the (finite) set of names of nodes in the interface of graph.*
2. G *is a term generated by the grammar*
   $G ::= nil \mid L(\boldsymbol{x}) \mid G_1|G_2 \mid \nu y\ G$
   where $\boldsymbol{x}$ *is a vector of names,* L *is an edge label with* $\mathrm{rank}(L) = |\boldsymbol{x}|$ *and* y *is a name.*

Here ν is a binder for names, thus we can define as usual functions $\mathrm{fn}(-)$, $\mathrm{bn}(-)$ and $\mathrm{n}(-)$ that compute the sets of free, bound and all the names respectively.

In the following, when no confusion will arise, we will use the terms names and nodes as synonyms in the setting of SHR.

(ax1) $G_1\|(G_2\|G_3) \equiv (G_1\|G_2)\|G_3$ (ax2) $G_1\|G_2 \equiv G_2\|G_1$ (ax3) $G\|nil \equiv G$
(ax4) $\nu x\ \nu y\ G \equiv \nu y\ \nu x\ G$ (ax5) $\nu x\ G \equiv G$ if $x \notin \mathrm{fn}(G)$
(ax6) $\nu x\ G \equiv \nu y\ G\{y/x\}$ if $y \notin \mathrm{fn}(G)$
(ax7) $\nu x\ (G_1\|G_2) \equiv (\nu x\ G_1)\|G_2$ if $x \notin \mathrm{fn}(G_2)$

Table 2. Structural congruence for graph terms.

Definition 9 (Structural congruence and well-formed judgements). *The structural congruence $\equiv$ on terms G obeys the axioms in Table 2. The well-formed judgements $\Gamma \vdash G$ over LE and $\mathcal{N}$ are those where* $\mathrm{fn}(G) = \Gamma$.

Well-formed judgements up to structural axioms are isomorphic to graphs up to isomorphisms. For a formal statement of the correspondence see [8].

We present now the steps of an SHR computation.

Definition 10 (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 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 $\boldsymbol{y}$ of nodes sent to x by the transition. If $\Lambda(x) = (a, \boldsymbol{y})$ we require that $\mathrm{ar}(a) = |\boldsymbol{y}|$ *and we define* $\mathrm{act}_\Lambda(x) = a$ *and* $\mathrm{n}_\Lambda(x) = \boldsymbol{y}$.

We also define:

- $\mathrm{n}(\Lambda) = \{z | \exists x.z \in \mathrm{n}_\Lambda(x)\}$
  set of communicated names;
- $\Gamma_\Lambda = \mathrm{n}(\Lambda) \setminus \Gamma$
  set of communicated fresh names;
- $\mathrm{n}(\pi) = \{x | \exists x' \neq x.x\pi = x'\pi\}$
  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 names of representatives can be communicated. Furthermore we require $\Phi = \Gamma\pi \cup \Gamma_\Lambda$, namely nodes are never erased and new nodes are bound unless transmitted.

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

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

We derive SHR transitions from basic productions that define the behaviour of single edges.

Definition 11 (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 w.r.t. names in $\{x_1, \ldots, x_n\} \cup \Phi$.

Note that productions specify the behaviour of a single edge, attached to distinct nodes. Transitions are created by composing productions for edges with distinct attachment nodes and then merging nodes. When nodes are merged, the actions on them must be synchronized. This is done according to a chosen *synchronization model*. For a general definition of synchronization models see [9]. In this work we concentrate on two particular synchronization models, the Hoare one and the Milner one. In the former, all the edges attached to a node are required to perform the same action, and this is forced by specifying that x and y can be merged only if $\mathrm{act}_\Lambda(x) = \mathrm{act}_\Lambda(y)$, and in that case the action performed on the resulting node is again $\mathrm{act}_\Lambda(x)$. In the latter, a pair of edges performs complementary actions (such as an input and an output) while the other edges attached to the same node perform a special idle action ϵ of arity 0 which stands for "no synchronization". The result of a Milner synchronization between two complementary actions is denoted with τ, while ϵ acts as neutral element for the synchronization.

In Milner model, just τ and ϵ are allowed on restricted nodes, while any action is allowed in the Hoare model.

In both the models, when synchronization is performed, the nodes in the tuples attached to matched actions are pairwise merged. The resulting fusion is applied to the label and to the resulting graph. Furthermore, the part dealing with nodes in Γ is recorded in the fusion π in the label.

Production composition is usually specified via inference rules. Since here we are more interested in the idea of SHR than in the technicalities, we refer to [10] for the inference rules for Hoare synchronization and to [11] for the ones for Milner synchronization.

As notational shortcut, when the action performed on a node is not specified, it is assumed to be ϵ.

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 [6] 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.

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 12. *We have for clauses (C) and goals (G) the following grammar:*

$$C ::= A \leftarrow G$$
$$G ::= G, G \mid A \mid \square$$

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

A logic program is a set of clauses. Derivations in logic programming are called SLD-derivations.

Definition 13 (SLD-derivation). *Let P be a logic program. We define a step of an SLD-resolution computation using the following rules:*

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

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.

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

We will omit $P \Vdash$ if P is clear from the context.
A SLD-derivation is a sequence (possibly empty) of steps of SLD-resolution.

3 Outline of the translation

We present here the different steps of the translation.

3.1 From Fusion Calculus to Milner SHR

The main point here is to translate a process calculus into a graph transformation framework. In process calculi, there is no clear distinction between the topological description of the system and the specification of its behaviour, which are both defined using term constructors. Some of them such as prefixes and non-deterministic sum are more "behavioural", while others such as parallel composition and restriction are more "topological".

When moving from process calculi to graphs, this distinction is made more explicit, since the topological part is represented in the graph structure, while the behavioural part is specified by productions (or, more in general, rewriting rules).

The idea of the translation is to map sequential processes (which have in our understanding no internal topological structure) into edges, which are the basic "computational" entities in graphs. Names, which represent communication infrastructure in process calculi, are mapped to nodes, which have the same role in

graph-based models. Restriction has the same meaning in both the cases, thus restricted names can be mapped into restricted nodes.

The label of an edge is computed from the sequential process it represents: since names are dealt with by nodes, the edge label contains the process, but with placeholders instead of occurrences of names. The real name corresponding to placeholder n is represented by the n-th node to which the edge is attached. This approach has the advantage of requiring a finite number of edge labels and productions for all the derivatives of a fixed process (this holds also for recursive processes, which may have an infinite number of derivatives). Note however that the labels used for edges are chosen just to make the correspondence clear: the choice of the label is immaterial, provided that this is used consistently and that the productions for edges with that label are the correct ones. In other terms, all the labels can be bijectively renamed without any problem, since behaviour is specified by productions and not by them.

At this point, productions have to be defined. As said, they are deduced from the process term too. In particular, we have a production for each prefix at the top level of a sequential process. We use in SHR actions in_n for input with n arguments and out_n for the output (and these are complementary actions). Thus an action prefix is executed by a production that produces the corresponding action on the subject node. Similarly, a fusion prefix ϕ is executed by a production that applies to the graph a substitution $\pi = \mathrm{mgu}(\phi)$. The result of the production is in both the cases the translation of the part of the process that follows the executed prefix. Note that this can be any graph (not only a single edge).

A detailed definition of this mapping (with standard names used as placeholders) can be found in [12].

Instead of showing the details of the mapping we present here a simple example.

Example 1. Let us consider the following process:

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

Even if this part of the translation can deal also with open processes, we will use here a closed process since we want to use this example as running example throughout the paper, and next step will work only with closed processes.

We have here three sequential processes, thus the corresponding graph has three edges. The result of the translation (in the textual representation) is:

$$\vdash (uxyzw)(L_{Q(1,2,3)}(x,y,z)|L_{\overline{1}23.R(4,5)}(u,x,y,u,x)|L_{123.S(4,5)}(u,z,w,z,w))$$

As far as productions are concerned, since we have two executable prefixes, we also have two productions:

$$u_1,x_1,y_1,u_2,x_2 \vdash L_{\overline{1}23.R(4,5)}(u_1,x_1,y_1,u_2,x_2) \xrightarrow{(u_1,out_2,\langle x_1,y_1\rangle)} u_1,x_1,y_1,u_2,x_2 \vdash L_{R(1,2)}(u_2,x_2)$$

$$u_1, z_1, w_1, z_2, w_2 \vdash L_{123.S(4,5)}(u_1, z_1, w_1, z_2, w_2) \xrightarrow{(u_1, in_2, \langle z_1, w_1 \rangle)} u_1, z_1, w_1, z_2, w_2 \vdash L_{S(1,2)}(z_2, w_2)$$

In addition, for each edge, we have a production specifying that it can stay idle. In particular, in the example we use one such production for edge $L_{Q(1,2,3)}$:

$$x_1, y_1, z_1 \vdash L_{Q(1,2,3)}(x_1, y_1, z_1) \rightarrow x_1, y_1, z_1 \vdash L_{Q(1,2,3)}(x_1, y_1, z_1)$$

The three productions can be combined to have the following transition because the two actions on u are complementary and thus can synchronize according to the Milner model, while on the other nodes just ϵ actions are executed.

$$\vdash (uxyzw)(L_{Q(1,2,3)}(x, y, z) | L_{\overline{1}23.R(4,5)}(u, x, y, u, x) | L_{123.S(4,5)}(u, z, w, z, w)) \rightarrow \vdash (uxy)(L_{Q(1,2,3)}(x, y, x) | L_{R(1,2)}(u, x) | L_{S(1,2)}(x, y))$$

This corresponds to the Fusion transition:

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

Here no label is visible since all the nodes are restricted. If, e.g., u would be free, then the label on it would be $(u, \tau, \langle\rangle)$. Note that in that case also other actions would be allowed on u, e.g. the output, while they are forbidden here because of restriction.

In general, a full correspondence can be found between the labelled transition system of Fusion Calculus and the interleaving part of the labelled transition system of Milner SHR, when translations of Fusion processes are used as starting graphs. Notice that in Fusion Calculus the semantics is interleaving, i.e., at each step we have just one action, which can be either an asynchronous action or a synchronization of two communication actions, yielding a fusion. In SHR, instead, the system is distributed, and there is no central control, thus different actions can be performed independently at the same time (i.e., in the same transition) but in different places (i.e., on different channels).

We conclude this part with some general considerations on the mapping. First of all, note that the mapping is extremely simple and, in particular, it is a uniform encoding (i.e., it preserves parallel composition and commutes with name substitutions) according to [13]. In addition to that, the semantic correspondence is very strong.

All that allows to consider Milner SHR as a generalization of Fusion Calculus with different allowed actions and with multiple synchronization, since a production can enforce many synchronization constraints at the same time, one for each adjacent node. Using Fusion-like syntax, the first extension corresponds to have prefixes of the form $xa\boldsymbol{y}$ where a is an arbitrary action, instead of just inputs and outputs. The second one instead corresponds to have parallel prefixes of the form, e.g., $(xy|\overline{y}z).P$. In that case both the prefixes must be executed at

the same time. Note that this form of multiple synchronization is quite powerful since it allows synchronization among any number of processes in a synchronous way (double prefix is enough since then synchronizations can be concatenated). We think that this kind of synchronization is an useful abstraction for modelling complex protocols/transactions in a simple way.

3.2 Changing the synchronization model

This step of the translation is internal to the SHR framework: we want to change the synchronization policy used from the Milner one to the Hoare one, since this last corresponds to logic programming unification.

The main differences between the Hoare and the Milner models are the following:

- in the Milner model, one pair of edges interacts at each step, while other edges possibly attached to the same node must stay idle whereas in the Hoare model all the edges connected to a node must interact;
- in the Milner model there is an asymmetry in the synchronization: there are one sender and one receiver, while Hoare synchronization is symmetric.

The first attempt to build a translation aims at changing just the SHR "program", i.e. the productions, while preserving the data, i.e. the graph. However it is easy to see that no translation of this kind can preserve the behaviour. Furthermore, no uniform encoding [13] can exist. Notice in fact that Hoare synchronization preserves symmetries in the graph, while Milner synchronization does not. For instance, let us consider $x \vdash C(x)|C(x)$ as starting graph. Using Hoare synchronization, with any set of productions, if we have a non idle transition, then we also have a non idle transition to a graph of the form $\Gamma \vdash G|G$ for some set of nodes Γ and some graph term G. At the contrary, with Milner synchronization, the productions $x \vdash C(x) \xrightarrow{(x,a,\langle\rangle)} x \vdash C(x)$ and $x \vdash C(x) \xrightarrow{(x,\overline{a},\langle\rangle)} x \vdash D(x)$ allow just one non idle transition which has $x \vdash C(x)|D(x)$ as final graph. In this case symmetry has clearly been broken. A full proof of the impossibility result can be produced using the techniques in [14] and in [13].

Thus, in order to implement Milner synchronization in the Hoare model we have to use a more complex mapping, and in particular we must change also the graph structure. We choose to substitute Milner nodes with complex networks implementing them. More in detail, a node which is attached to n tentacles of edges becomes a net, called *amoeboid*, with n external nodes, each of them attached to one of these tentacles.

Amoeboids behave as nodes, and in particular they must satisfy the two following properties:

1. they must allow synchronization only if two complementary actions are provided from the outside;
2. they must merge the amoeboids corresponding to transmitted names.

An easy implementation that satisfies the first condition uses a class of edges am_k of arity k, with productions:

$$\Gamma \vdash am_k(\Gamma) \xrightarrow{(x_1,in_n,\boldsymbol{y}),(x_2,out_n,\boldsymbol{y})} \Gamma, \boldsymbol{y} \vdash am_k(\Gamma)$$

where Γ is a set of nodes such that $|\Gamma| = k$, x_1 and x_2 are arbitrary nodes in Γ and $\boldsymbol{y}$ is a vector of fresh names such that $|\boldsymbol{y}| = n$. Note that such a production must be provided for each k and n and for each choice of x_1 and x_2 in Γ.

This realization satisfies the first condition, but it does not allow to satisfy the second one. In fact, fusions are applied by merging nodes (thus the result will use Hoare synchronization), not by merging amoeboids.

Since there is no primitive notion of merging of edges, and implementing this primitive is very difficult (since we have to deal with concurrent merges triggered by independent processes), we decide to generalize the concept of amoeboid. We will use as amoeboids any network with a suitable structure. In particular:

- amoeboids are composed by particular kinds of edges (identified by their labels);
- amoeboids have an interface, that is a set of nodes corresponding to the tentacles attached to the represented Milner node;
- each internal node in an amoeboid is shared by exactly two edges, each external node is attached to one edge;
- connecting two amoeboids using a third one corresponds to merge the two amoeboids into a single one.

In particular, amoeboids can be implemented using two kinds of edges, am_k edges acting as routers as before, but that create new amoeboids instead of merging nodes, and *null* edges of arity 1 which are used to close disconnected elements in the interface by forbidding any synchronization on them.

As important invariant, each node in the translated graph is shared by exactly two edges. This is intuitively a useful property since in this case the first difference between Hoare and Milner synchronization is immaterial.

Productions for am_k edges are described below, while no productions are available for *null* edges. As notation, we use $\prod_{i\in I} G_i$ to denote the parallel composition of a family of graphs G_i indexed by the elements $i \in I$.

Definition 14 (Auxiliary productions). *We have auxiliary productions of the form:*

$$\Gamma \vdash am_k(\Gamma) \xrightarrow{(x_1,in_n,\boldsymbol{y}_1),(x_2,out_n,\boldsymbol{y}_2)} \Gamma, \boldsymbol{y}_1, \boldsymbol{y}_2 \vdash am_k(\Gamma) \mid \prod_{i=1\ldots|\boldsymbol{y}_1|} am_2(\boldsymbol{y}_1[i], \boldsymbol{y}_2[i])$$

where Γ is a chosen tuple of distinct names with k components and $\boldsymbol{y}_1$ and $\boldsymbol{y}_2$ are two vectors of fresh names such that $|\boldsymbol{y}_1| = |\boldsymbol{y}_2| = n$. We need such a production for each k and n and each pair of nodes x_1 and x_2 in Γ.

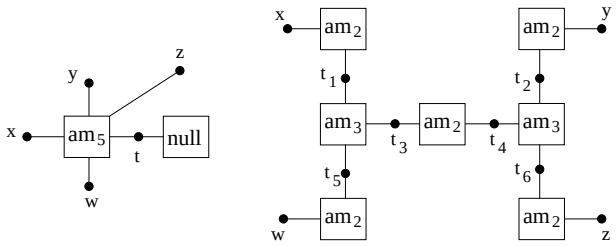

Fig. 1. Two sample amoeboids.

Note that each edge in the amoeboid flips the polarity of the synchronization from in_n to out_n and viceversa, thus in order to have opposite polarities at the two ends we require that each path between elements of the interface has odd length. Figure 1 shows two sample amoeboids with the same interface x, y, w, z.

Productions for process edges have to be modified in order to preserve these properties and in order to close nodes which are no more used with *null* edges. The technical definition of productions is a bit complex and we refer to [7] for its presentation.

We just present here the second step of the translation for our running example.

Example 2. We have as graph in the Milner setting:

$$\vdash (uxyzw)(L_{Q(1,2,3)}(x,y,z)|L_{\overline{1}23.R(4,5)}(u,x,y,u,x)|L_{123.S(4,5)}(u,z,w,z,w))$$

We can translate it using for each amoeboid the simplest possible form, i.e. a single *am* edge, obtaining:

$$\begin{aligned}\vdash\ & (u_1u_2u_3x_1x_2x_3y_1y_2z_1z_2z_3w_1w_2)\\ & (L_{Q(1,2,3)}(x_1,y_1,z_1)|L_{\overline{1}23.R(4,5)}(u_1,x_2,y_2,u_2,x_3)|L_{123.S(4,5)}(u_3,z_2,w_1,z_3,w_2)|\\ & am_3(x_1,x_2,x_3)|am_2(y_1,y_2)|am_3(z_1,z_2,z_3)|am_3(u_1,u_2,u_3)|am_2(w_1,w_2))\end{aligned}$$

As far as productions are concerned, we have to translate the following two productions:

$$u_1,x_1,y_1,u_2,x_2 \vdash L_{\overline{1}23.R(4,5)}(u_1,x_1,y_1,u_2,x_2) \xrightarrow{(u_1,out_2,\langle x_1,y_1\rangle)} u_1,x_1,y_1,u_2,x_2 \vdash L_{R(1,2)}(u_2,x_2)$$

$$u_1,z_1,w_1,z_2,w_2 \vdash L_{123.S(4,5)}(u_1,z_1,w_1,z_2,w_2) \xrightarrow{(u_1,in_2,\langle z_1,w_1\rangle)} u_1,z_1,w_1,z_2,w_2 \vdash L_{S(1,2)}(z_2,w_2)$$

The result is:

$$u_1, x_1, y_1, u_2, x_2 \vdash L_{\overline{1}23.R(4,5)}(u_1, x_1, y_1, u_2, x_2) \xrightarrow{(u_1, out_2, \langle x_1, y_1 \rangle)} u_1, x_1, y_1, u_2, x_2 \vdash L_{R(1,2)}(u_2, x_2) | null(u_1)$$

$$u_1, z_1, w_1, z_2, w_2 \vdash L_{123.S(4,5)}(u_1, z_1, w_1, z_2, w_2) \xrightarrow{(u_1, in_2, \langle z_1, w_1 \rangle)} u_1, z_1, w_1, z_2, w_2 \vdash L_{S(1,2)}(z_2, w_2) | null(u_1)$$

Notice that, in the first production, u_1 is closed using a *null* edge since it is no more used, while x_1 and y_1 are not, since they are communicated and thus will be used by other edges.

These two productions can be combined with the following auxiliary production for the amoeboid for u:

$$u_1, u_2, u_3 \vdash am_3(u_1, u_2, u_3) \xrightarrow{(u_1, out_2, \langle t_1, t_2 \rangle), (u_3, in_2, \langle t_3, t_4 \rangle)} u_1, u_2, u_3, t_1, t_2, t_3, t_4 \vdash am_3(u_1, u_2, u_3) | am_2(t_1, t_3) | am_2(t_2, t_4)$$

With these productions and the idle productions for other edges we derive the following transition:

$$\begin{gathered} \vdash (u_1 u_2 u_3 x_1 x_2 x_3 y_1 y_2 z_1 z_2 z_3 w_1 w_2) \\ (L_{Q(1,2,3)}(x_1, y_1, z_1) | L_{\overline{1}23.R(4,5)}(u_1, x_2, y_2, u_2, x_3) | L_{123.S(4,5)}(u_3, z_2, w_1, z_3, w_2) | \\ am_3(x_1, x_2, x_3) | am_2(y_1, y_2) | am_3(z_1, z_2, z_3) | am_3(u_1, u_2, u_3) | am_2(w_1, w_2)) \\ \rightarrow \\ \vdash (u_1 u_2 u_3 x_1 x_2 x_3 y_1 y_2 z_1 z_2 z_3 w_1 w_2) \\ (L_{Q(1,2,3)}(x_1, y_1, z_1) | L_{R(1,2)}(u_2, x_3) | null(u_1) | L_{S(1,2)}(z_3, w_2) | null(u_3) | \\ am_3(x_1, x_2, x_3) | am_2(y_1, y_2) | am_3(z_1, z_2, z_3) | \\ am_3(u_1, u_2, u_3) | am_2(x_2, z_2) | am_2(y_2, w_1) | am_2(w_1, w_2)) \end{gathered}$$

Note that now the amoeboids $am_2(y_1, y_2)$ and $am_2(w_1, w_2)$ have been connected through the amoeboid $am_2(y_2, w_1)$, thus there is now just one amoeboid $am_2(y_1, y_2) | am_2(y_2, w_1) | am_2(w_1, w_2)$, and this corresponds to having merged nodes y and w. Also note that the final graph satisfies the invariant of having each node shared by exactly two edges and that each path between external nodes of an amoeboid, e.g., the path $am_2(y_1, y_2)$, $am_2(y_2, w_1)$, $am_2(w_1, w_2)$ between nodes y_1 and w_2, has odd length.

The correspondence between the two models can be found only for closed processes. This happens because nodes are represented by complex distributed structures: suppose to add to amoeboids some information recording whether they are free or not. When they are merged, a restricted amoeboid may become part of a free one, so it must change its behaviour. But since the merge does not

involve any synchronization inside the amoeboid, this cannot be done. Adding a synchronization for that is quite complex since many concurrent merges can happen.

Note also that there is no way of forcing atomicity inside amoeboids (this can be proved by considering a situation with two symmetric synchronizations on a symmetric amoeboid), in the sense that a distributed amoeboid may allow two distinct, disjoint synchronizations at the same time. While this is a problem for a general translation from Milner SHR into Hoare SHR, when considering just translations of Fusion Calculus processes the same interleaving condition required to have a faithful correspondence with Fusion Calculus also guarantees that there are never more than two available synchronizations, thus atomicity comes for free. When looking for a more concurrent semantics, in the Milner model we just have to drop the interleaving condition and consider the semantics induced by the SHR framework, which forces interleaving on each channel, while allowing concurrency among channels. In the Hoare model instead, different amoeboids for the same set of nodes which are equivalent in the interleaving scenario are no more equivalent in a concurrent one. In particular, concurrent synchronizations are allowed when they can be performed using disjoint paths inside the amoeboid to connect the corresponding actions. For instance, in the second amoeboid of Figure 1, we can have at the same time a synchronization between x and w and another one between y and z, while in the first amoeboid this is not allowed.

The mapping not only has those limitations, but it is also quite complex. We argue that this is due to the fact that the two selected synchronization models, even if quite simple, are far away one from the other. This observation, together with the observation that in real systems one can have complex synchronization models, or may want to abstract whole protocols as synchronization primitives, triggered the idea of modeling the synchronization model as a separate entity, which can be modified independently w.r.t. the used framework. This idea was formalized using an extension of synchronization algebras [15], called synchronization algebras with mobility, and was successfully applied both in the framework of SHR [9,16] and in the framework of process calculi [17].

As final remark, we can notice that in the Hoare model restriction is just "observational", in the sense that no transition can be added or forbidden by restrictions, but just labels are changed. Since here we are not interested in the labels but just in the graph, restrictions can be dropped.

3.3 From Hoare SHR to logic programming

The final step of our translation is from Hoare SHR to logic programming. The idea of the mapping is to map graphs into a particular kind of goals and to use clauses to rewrite them. The mapping is essentially an isomorphism (apart from the Γ part, which is dropped) with edges attached to nodes mapped into predicates applied to variables, and parallel composition matched by AND composition.

A main difference between the two models is that while in SHR productions are synchronized to form transitions, in logic programming clauses are applied in sequence instead of at the same time, and in no particular order. To solve this mismatch we have to introduce a transaction mechanism also in logic programming. We call Synchronized Logic Programming the resulting paradigm. For more details on it see [10]. Transactions are introduced by using a standard technique used, e.g., in zero-safe nets [18].

We define a subset of the states (i.e. of the goals) as stable and we force transactions to go from a stable state to another stable state. More in particular, a goal is stable iff it does not contain function (or constant) symbols. We call such a goal goal-graph since it can be obtained as result of the translation of a graph. We call big-step a transaction, i.e. a sequence of logic programming steps from a goal-graph to another goal-graph.

Function symbols are used to model synchronization: when unification makes a function symbol f to appear in the goal, the big-step cannot end until another step makes f to disappear. Suppose that an atom $p(x)$ is unified with the head of a clause of the form $p(f(\dots))$. This corresponds to execute action f at x. When the mgu is applied to the goal, each occurrence of x is substituted by $f(\dots)$. In order to get rid of it, the predicate containing it must be unified with $f(\dots)$, and this must happen for all the occurrences of x in the starting goal. Thus all the predicates containing x must be unified with $f(\dots)$, and this corresponds in the SHR setting to the constraint that each edge attached to node x must execute action f on it, i.e. to Hoare synchronization.

The transaction mechanism as described so far is quite general, but we have to introduce some particular conditions in order to capture the specific kind of transactions that are used in SHR. There we have that in each transition an edge performs one action on each channel (we have no structured actions), that newly created edges cannot act before the next transition and that an edge is rewritten into a graph with no pending synchronization constraints.

This amounts to say that in logic clauses we have no nested function symbols, that predicates introduced in a big-step cannot be rewritten in the same big-step and that the body of each clause is a goal-graph.

In order to complete the translation we have still one issue to consider: name handling. First, fusions are performed by unification, with the nodes attached to each action modeled as arguments of the function symbol modeling it. Note however that, in SHR, when we execute an action f on node x, we do not "consume" node x. Remember that in logic programming this is performed by unifying x with a term of the form $f(\dots)$. Thus, since substitutions are idempotent, x will disappear from the goal. We have thus to find another representation for x. We decide to add a new argument to f (in the first position by convention), which will be the new name for x. Thus the translation of a SHR node in the logic programming setting is a chain of variables, linked since each variable in the chain is the first argument of the term unified with its predecessor.

A final consideration is necessary for action ϵ. In [7,10], the translation dropped action ϵ by using simply x instead of $\epsilon(x)$. This was done to have a

less synchronous semantics (since this avoids chains of ϵ synchronizations among independent part of the same transition) and to avoid to translate idle productions which become unnecessary. However in some cases (when a variable with no synchronizations on it in the head of a clause does not appear in the body of the same clause) ϵ has to be reintroduced in order to avoid wrong transactions. Since this case is also quite frequent when translating Fusion processes (e.g., it occurs in our running example) and since using explicit ϵ makes the correspondence more straightforward, we have decided to treat it as a normal action.

As usual we skip the technicalities of the translation and present just its application to our running example.

Example 3. We continue here the running example, but for simplicity (and to make synchronization visible) we discard restrictions. This can be done since, as already said, restriction just hides labels. We also drop the Γ part since it is not translated (this amounts to work up to isolated nodes).

Remember that the starting graph is:

$$L_{Q(1,2,3)}(x_1,y_1,z_1)|L_{\overline{1}23.R(4,5)}(u_1,x_2,y_2,u_2,x_3)|L_{123.S(4,5)}(u_3,z_2,w_1,z_3,w_2)|$$
$$am_3(x_1,x_2,x_3)|am_2(y_1,y_2)|am_3(z_1,z_2,z_3)|am_3(u_1,u_2,u_3)|am_2(w_1,w_2)$$

which is translated into the goal:

$$L_{Q(1,2,3)}(x_1,y_1,z_1), L_{\overline{1}23.R(4,5)}(u_1,x_2,y_2,u_2,x_3), L_{123.S(4,5)}(u_3,z_2,w_1,z_3,w_2),$$
$$am_3(x_1,x_2,x_3), am_2(y_1,y_2), am_3(z_1,z_2,z_3), am_3(u_1,u_2,u_3), am_2(w_1,w_2)$$

We also have to translate the three productions:

$$L_{\overline{1}23.R(4,5)}(u_1,x_1,y_1,u_2,x_2) \xrightarrow{(u_1,out_2,\langle x_1,y_1\rangle)} L_{R(1,2)}(u_2,x_2)|null(u_1)$$

$$L_{123.S(4,5)}(u_1,z_1,w_1,z_2,w_2) \xrightarrow{(u_1,in_2,\langle z_1,w_1\rangle)} L_{S(1,2)}(z_2,w_2)|null(u_1)$$

$$am_3(u_1,u_2,u_3) \xrightarrow{(u_1,out_2,\langle t_1,t_2\rangle),(u_3,in_2,\langle t_3,t_4\rangle)} am_3(u_1,u_2,u_3)|am_2(t_1,t_3)|am_2(t_2,t_4)$$

into the corresponding clauses:

$$L_{\overline{1}23.R(4,5)}(out_2(u_1,x_1,y_1),\epsilon(x_1),\epsilon(y_1),\epsilon(u_2),\epsilon(x_2)) \leftarrow L_{R(1,2)}(u_2,x_2),null(u_1)$$

$$L_{123.S(4,5)}(in_2(u_1,z_1,w_1),\epsilon(z_1),\epsilon(w_1),\epsilon(z_2),\epsilon(w_2)) \leftarrow L_{S(1,2)}(z_2,w_2),null(u_1)$$

$$L_{Q(1,2,3)}(x_1,y_1,z_1), L_{\overline{123}.R(4,5)}(u_1,x_2,y_2,u_2,x_3), L_{123.S(4,5)}(u_3,z_2,w_1,z_3,w_2),$$
$$am_3(x_1,x_2,x_3), am_2(y_1,y_2), am_3(z_1,z_2,z_3), am_3(u_1,u_2,u_3), am_2(w_1,w_2)$$
$$\xrightarrow{out_2(u_1',x_2',y_2')/u_1,\epsilon(x_2')/x_2,\epsilon(y_2')/y_2,\epsilon(u_2')/u_2,\epsilon(x_3')/x_3}$$
$$L_{Q(1,2,3)}(x_1,y_1,z_1), L_{R(1,2)}(u_2',x_3'), null(u_1'), L_{123.S(4,5)}(u_3,z_2,w_1,z_3,w_2),$$
$$am_3(x_1,\epsilon(x_2'),\epsilon(x_3')), am_2(y_1,\epsilon(y_2')), am_3(z_1,z_2,z_3),$$
$$am_3(out_2(u_1',x_2',y_2'),\epsilon(u_2'),u_3), am_2(w_1,w_2)$$
$$\xrightarrow{u_1'/u_1'',x_2'/t_1,y_2'/t_2,u_2'/u_2'',in_2(u_3',t_3,t_4)/u_3}$$
$$L_{Q(1,2,3)}(x_1,y_1,z_1), L_{R(1,2)}(u_2',x_3'), null(u_1'), L_{123.S(4,5)}(in_2(u_3',t_3,t_4),z_2,w_1,z_3,w_2),$$
$$am_3(x_1,\epsilon(x_2'),\epsilon(x_3')), am_2(y_1,\epsilon(y_2')), am_3(z_1,z_2,z_3),$$
$$am_3(u_1',u_2',u_3'), am_2(x_2',t_3), am_2(y_2',t_4), am_2(w_1,w_2)$$
$$\xrightarrow{u_3'/u_3'',z_2'/t_3,\epsilon(z_2')/z_2,w_1'/t_4,\epsilon(w_1')/w_1,\epsilon(z_3')/z_3,\epsilon(w_2')/w_2}$$
$$L_{Q(1,2,3)}(x_1,y_1,z_1), L_{R(1,2)}(u_2',x_3'), null(u_1'), L_{S(1,2)}(z_3',w_2'), null(u_3'),$$
$$am_3(x_1,\epsilon(x_2'),\epsilon(x_3')), am_2(y_1,\epsilon(y_2')), am_3(z_1,\epsilon(z_2'),\epsilon(z_3')),$$
$$am_3(u_1',u_2',u_3'), am_2(x_2',z_2'), am_2(y_2',w_1'), am_2(\epsilon(w_1'),\epsilon(w_2'))$$
$$\xrightarrow{\cdots}$$
$$\cdots \quad \cdots \quad \cdots$$
$$\xrightarrow{\cdots}$$
$$L_{Q(1,2,3)}(x_1',y_1',z_1'), L_{R(1,2)}(u_2',x_3'), null(u_1'), L_{S(1,2)}(in_2(z_3',w_2'), null(u_3'),$$
$$am_3(x_1',x_2',x_3'), am_2(y_1',y_2'), am_3(z_1',z_2',z_3'),$$
$$am_3(u_1',u_2',u_3'), am_2(x_2',z_2'), am_2(y_2',w_1'), am_2(w_1',w_2')$$

Table 3. The big-step of the running example.

$$am_3(out_2(u_1,t_1,t_2),\epsilon(u_2),in_2(u_3,t_3,t_4)) \leftarrow$$
$$am_3(u_1,u_2,u_3), am_2(t_1,t_3), am_2(t_2,t_4)$$

Using their suitably renamed variants and all the translations of the idle productions we can have the SLD-computation in Table 3. Note that the last clauses just perform ϵ synchronizations. The final goal is the translation of the final graph of the SHR transition, but now we have new variables, e.g. x_1' instead of x_1. Note also that by considering the composition of the mgus, restricted to the variables in the starting goal, we have exactly Λ. For instance, the substitution $out_2(u_1',x_1',y_1')/u_1$ proves that action out_2 has been performed on u_1 with arguments x_1 and y_1 (represented by their successors x_1' and y_1') and that the successor of u_1 is u_1'.

As for the translation from Fusion Calculus to Milner SHR, we have here a simple mapping (even if it is not a uniform encoding since nodes become sequences of variables) together with a full semantic correspondence, and this

allows to consider Hoare SHR as a particular case of Synchronized Logic Programming.

In fact, it is easy to extend the correspondence by taking into account also SHR restriction and logic programming constants and nested functions. As far as restriction is concerned, the same feature can be added to logic programming. Interestingly, it amounts to perform in a step by step way the classical operation of restricting the computed answer substitution to the (free) variables of the starting goal. In the other direction, constants are simply actions that destroy the node where they are performed, while nested functions are structured synchronization patterns. As natural, to use the full power of these generalized synchronizations, a more flexible transactional mechanism must be used (i.e. actions of one edge must be matched by sequences of actions from other edges).

A more interesting but also more challenging extension is to have non Hoare logic programming. We argue that this requires to have a different unification algorithm, but also substitutions and goals have to be managed differently. In fact, the quite strong mathematical properties of Hoare unification makes its asynchronous version, i.e. standard unification, quite simple. For other synchronizations models, it may be necessary to keep some more information in the state. In particular, when a variable is substituted with a function symbol, it may be useful to preserve the name of the variable in the other predicates where it occurs for later use.

4 Conclusions

We have presented here a chain of mappings that allows to link different formalisms for modeling interactive distributed systems such as the ones used in `GC`. We can now give a new description of them that is the result of what we learned from the mappings.

- Fusion Calculus is a process calculus with an interleaving semantics based on the Milner synchronization model.
- SHR is a distributed model, which thus has a concurrent semantics, in the sense that different actions can happen in different places inside the same transition. It can be seen as a generalization of Fusion Calculus.
- Logic programming is an asynchronous goal-rewriting framework which is based on "Hoare" unification. A transaction mechanism must be added to it in order to enforce real synchronization.

Many insights on the analyzed models and proposals for further extensions have been suggested at the end of the various sections. We collect however here the main ones.

A main point comes out by comparing the three mappings: the one between Hoare SHR and Milner SHR is by far the most complex. This suggests that in general it is not easy to implement a synchronization model, even if it is quite simple like Hoare and Milner ones are, with the primitives provided by another one. This observation, together with the fact that more complex synchronization

models may be desirable, maybe as abstractions of complex protocols of real systems, triggered the idea of having the synchronization model as a parameter of the modeling framework. We have already worked on that part, both in the setting of SHR [9,16] and in the setting of process calculi [17], but a lot of work is still to do. In particular, the expressive power of the parametric models thus obtained must be analyzed, but this is not an easy task since the expressiveness depends on the choice of the synchronization model.

The synchronization model can be changed also in logic programming, but here there is the problem of finding a different unification algorithm which is the asynchronous counterpart of a general synchronization model, in the same way as usual unification is the counterpart of Hoare synchronization.

Furthermore, Synchronized Logic Programming, which has been used here mainly to have a counterpart of SHR synchronization in the logic programming framework, deserves further analysis. In particular, it should be compared with other models with transactions in order to fully understand its expressivity.

As final word, we think that our comparisons have provided many insights on the analyzed models and many suggestions concerning how to extend them. For this reason, it would be useful to complete the work in order to have a coherent picture of all the existing proposals for models for `GC` systems. Naturally, seen the abundance of proposals in the literature, this is a long and hard work.

References

1. Victor, B.: The Fusion Calculus: Expressiveness and Symmetry in Mobile Processes. PhD thesis, Dept. of Computer Systems, Uppsala University, Sweden (1998)
2. Parrow, J., Victor, B.: The fusion calculus: Expressiveness and symmetry in mobile processes. In: Proc. of LICS '98, IEEE, Computer Society Press (1998)
3. Milner, R., Parrow, J., Walker, J.: A calculus of mobile processes, I and II. Inform. and Comput. **100** (1992) 1–40,41–77
4. Degano, P., Montanari, U.: A model for distributed systems based on graph rewriting. Journal of the ACM **34** (1987) 411–449
5. Hirsch, D., Montanari, U.: Synchronized hyperedge replacement with name mobility. In: Proc. of CONCUR'01. Volume 2154 of Lect. Notes in Comput. Sci., Springer (2001) 121–136
6. Lloyd, J.: Foundations of Logic Programming. Springer (1987)
7. Lanese, I., Montanari, U.: Mapping fusion and synchronized hyperedge replacement into logic programming. Theory and Practice of Logic Programming, Special Issue on Multiparadigm Languages and Constraint Programming (2004) To appear.
8. Hirsch, D.: Graph Transformation Models for Software Architecture Styles. PhD thesis, Departamento de Computación, Facultad de Ciencias Exactas y Naturales, U.B.A. (2003)
9. Lanese, I., Montanari, U.: Synchronization algebras with mobility for graph transformations. In: Proc. of FGUC'04. Elect. Notes in Th. Comput. Sci. (2004) To appear.
10. Lanese, I.: Process synchronization in distributed systems via Horn clauses. Master's thesis, University of Pisa, Computer Science Department (2002) Downloadable from `http://www.di.unipi.it/~lanese/work/tesi.ps`.

11. Ferrari, G., Montanari, U., Tuosto, E.: A LTS semantics of ambients via graph synchronization with mobility. In: Proc. of ICTCS'01. Volume 2202 of Lect. Notes in Comput. Sci., Springer (2001) 1–16
12. Lanese, I., Montanari, U.: A graphical fusion calculus. In: Proceedings of the Workshop of the COMETA Project on Computational Metamodels. Volume 104 of Elect. Notes in Th. Comput. Sci., Elsevier (2004) 199–215
13. Palamidessi, C.: Comparing the expressive power of the synchronous and the asynchronous pi-calculus. In: Proc. of POPL'97, ACM Press (1997) 256–265
14. Lynch, N.A.: A hundred impossibility proofs for distributed computing. In: Proc. of PODC'89, ACM Press (1989) 1–28
15. Winskel, G.: Synchronization trees. Theoret. Comput. Sci. **34** (1984) 33–82
16. Lanese, I., Tuosto, E.: Synchronized hyperedge replacement for heterogeneous systems. In: Proc. of COORDINATION 2005. Volume 3454 of Lect. Notes in Comput. Sci., Springer (2005) 220–235 To appear.
17. Bruni, R., Lanese, I.: PRISMA: A parametric calculus based on synchronization algebras with mobility. In: Proc. of CONCUR'05, Springer (2005) Submitted.
18. Bruni, R., Montanari, U.: Zero-safe nets: Comparing the collective and individual token approaches. Inform. and Comput. **156** (2000) 46–89