Available online at www.sciencedirect.com

SCIENCE DIRECT

Electronic Notes in Theoretical Computer Science

ELSEVIER Electronic Notes in Theoretical Computer Science 117 (2005) 335–351

www.elsevier.com/locate/entcs

Ambient Graph Rewriting

Pietro Cenciarelli, Ivano Talamo, and Alessandro Tiberi

Dipartimento di Informatica, University of Rome - "La Sapienza", Italy

Abstract

We investigate *synchronized hyperedge replacement* (SHR) as general framework for distributed programming and system design. We propose a slender version of SHR which dramaticaly reduces the mathematical overhead of the original proposal [5] and use it to interpret the *distributed CCS* [7] and the calculus of *Mobile Ambients* [1] in a uniform semantic framework. The encodings are *bisimulations*. A tool for supporting distributed system design and analysis is presented. The tool, which adopts the slender SHR as intermediate language, integrates model checking techniques within the framework of (distributed) program development.

Keywords: synchronized hyperedge replacement, graph rewriting, Mobile Ambients, distributed CCS, model checking.

1 Introduction

Synchronized hyperedge replacement, *SHR* [5], is a graph rewrite system for modelling process interaction in a network environment. In this framework, inspired by [4], hyperedges are to represent agents, or software components, while nodes are thought of as channels, synchronisation points or, more generally, network communication infrastructure. In [5] SHR has been used to provide a labelled transition system semantics of the calculus of *Mobile Ambients* [1].

The idea that hypergraphs may interact by synchronising action and co-action pairs at specific synchronisation points (the nodes) is quite intuitive, while the flexibility of the model in representing diverse network topologies and communication protocols makes SHR a reasonable candidate as common semantic framework for interpreting different calculi. Unfortunately, the admittedly considerable mathematical overhead involved in the original proposal

1571-0661/$ – see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.entcs.2004.06.017

tends to obscure the basic intuition and makes it hard to prove metatheoretical properties.

In the present paper we propose a slender version of SHR based on a single, rather intuitive rule of parallel composition. While drammatically reducing mathematical complexity we do not loose in expressive power. Two case studies are presented where the mobile ambients and DCCS, the *distributed* CCS of [7], are encoded in the proposed version of SHR. In particular, both models adopt a common recursive graph architecture whose components we call *ambient graphs*. Interpretation maps DCCS terms to *flat* ambient graphs, while mobile ambients are trees. Both encodings are proven to be *bisimulations* (thus improving the result obtained for ambients in [5]).

SHR is adopted as the intermediate language of the *Synchronised Hypergraph Environment* (*She*), a tool for system design and software development supporting concurrent and distributed programming. The system, which is currently being developed by the authors, integrates the techniques of model checking (which is more ofted used for verifying fully developed systems) within the framework of (distributed) program development. In particular, *She* uses *Murphi* [3] to compute all possible state transitions of the user's program, and checks that user specified *invariants* hold at all reachable states. Specific traces can be selected and the corresponding sequence of graph rewriting can be visualised as an *animation*. The system provides counterexamples if states violating the invariants are reachable.

Synopsis.

In section 2 we describe our proposed version of SHR. The two case studies are developed in section 3 (DCCS) and section 4 (ambient). *She* is described in section 5.

Notation.

We write $\boldsymbol{x}$ for a finite sequence $x_1, x_2, \ldots x_n$. If $f : A \times B$ is a relation and $a \in A$, we write $f(a)$ the set $\{b \in B \mid (a, b) \in f\}$. The *domain* of f is $dom(f) = \{x \in A \mid \exists\, b \in B \,.\, (a, b) \in f\}$.

2 Synchronized Hyperedge Replacement

Let $\mathcal{N}$ and $\mathcal{L}$ be sets (which we consider fixed throughout) respectively of *nodes* and *labels*. A *graph* $G = (E, \lambda, G)$ consists of a set E of *hyperedges* (or just *edges*), a labelling function $\lambda : E \to \mathcal{L}$ and an attachment function $G : E \to \mathcal{N}^*$. When $G(e) = x_1 x_2 \ldots x_n$ we call n the *arity* of e and say that

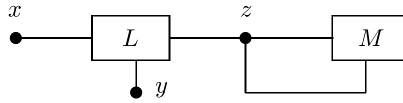


Table 1
$L(x, y, z) \,|\, M(z, z)$

the i-th *tentacle* of e is attached to x_i. The collection of nodes of a graph G is written:

$$|G| = \{x \in \mathcal{N} \mid \exists\, e \,.\, G(e) = x_1 \ldots x_n \text{ and } x = x_i\}.$$

When no confusion arises, we let $L(\boldsymbol{x})$ denote an edge e of a graph G with label $L \in \mathcal{L}$ and such that $G(e) = \boldsymbol{x}$. If G and H are graphs, we write $G|H$ the graph whose set of edges is the disjoint union of the edges of G and H, and whose labelling and attachment functions are the obvious ones. We picture graphs by drawing labelled boxes for edges and bullets for nodes; tentacles are represented by lines connecting the former to the latter. Table 1 shows a graph of the form $L(x, y, z) \,|\, M(z, z)$.

Let $Act = \{a, b, \ldots\} \cup \{\overline{a}, \overline{b}, \ldots\}$ be a set of *actions* and *co-actions* (overlined), and let $\overline{\overline{a}}$ denote a. We write Act^+ the set $Act \times \mathcal{N}^*$. Given $(a, \boldsymbol{x})$ in Act^+, we call the components of $\boldsymbol{x}$ *arguments* of a. A *pre-transition* of a graph G to a graph H, written:

$$G \xrightarrow{\Lambda} H,$$

is a relation $\Lambda \subseteq \mathcal{N} \times Act^+$ such that $dom(\Lambda) \subseteq |G|$. We write $(x, a, \boldsymbol{y})$ for an element $(x, (a, \boldsymbol{y}))$ of Λ, and (x, a) when $\boldsymbol{y}$ is the empty sequence. Intuitively, $(a, \boldsymbol{y}) \in \Lambda(x)$ expresses the occurrence of action a at node x. In SHR the occurence of both $(a, \boldsymbol{y})$ and $(\overline{a}, \boldsymbol{z})$ at x triggers a synchronisation between two agents (edges) of the graph, what is traditionally represented by a *silent* action τ. When such is the case the synchronising agents may exchange information. This is implemented in SHR by unifying the lists $\boldsymbol{y}$ and $\boldsymbol{z}$ of parameters, which are required to be of the same length. Only two agents at a time may synchronise at one node. Formally, let $\Lambda \subseteq \mathcal{N} \times Act^+$ be a relation, and let the expression $\upsilon(\Lambda)$ be either undefined or denote a node *substitution*, that is a partial function $\mathcal{N} \rightharpoonup \mathcal{N}$. In particular, let $\upsilon(\Lambda)$ be defined if and only if, for all $x \in \mathcal{N}$, the set $\Lambda(x)$ has at most two elements and, when so, it is $\{(a, \boldsymbol{y}), (\overline{a}, \boldsymbol{z})\}$, where the lengths of vectors $\boldsymbol{y}$ and $\boldsymbol{z}$ coincide. When $\upsilon(\Lambda)$ is defined, the substitution it denotes is the most general unifier of the arguments of all synchronised actions in Λ:

$$\upsilon(\Lambda) = mgu\ \{\boldsymbol{y} = \boldsymbol{z} \mid \exists\, x \,.\, \Lambda(x) = \{(a, \boldsymbol{y}), (\overline{a}, \boldsymbol{z})\}\}.$$

A *transition* is a pre-transition $G \xrightarrow{\Lambda} H$ such that $H = \rho(H)$, where $\rho = \upsilon(\Lambda)$. We say that an action a is *observed* at node x during a transition Λ if $\Lambda(x) = \{(a, \boldsymbol{y})\}$.

The rule [sync] below inplements synchronisation in SHR. It is subject to a *non-interference* condition: two transitions $G \xrightarrow{\Lambda} H$ and $F \xrightarrow{\Theta} K$ can be synchronised provided the nodes in Λ (the parameters) and in H which are new (do not appear) in G are also new in F, Θ and K; and vice-versa. Formally, let $|\Lambda|$ be the set $\{y \in \mathcal{N} \mid \exists x\,.\,(a, y_1 \dots y_n) \in \Lambda(x) \text{ and } y = y_i\}$; two transitions $G \xrightarrow{\Lambda} H$ and $F \xrightarrow{\Theta} K$ are said to be *non-interfering* if and only if:

- $x \in |K| \cup |\Theta|$ implies $x \in |F|$ or $x \notin |G| \cup |\Lambda| \cup |H|$, and
- $y \in |H| \cup |\Lambda|$ implies $y \in |G|$ or $y \notin |F| \cup |\Theta| \cup |K|$.

The rule for synchronisation is as follows:

$$[\,\text{sync}\,] \qquad \frac{G \xrightarrow{\Lambda} H \quad F \xrightarrow{\Theta} K}{G \,|\, F \xrightarrow{\Lambda \cup \Theta} \rho(H|K)} \qquad (\star)$$

$(\star)$ if $\rho = \upsilon(\Lambda \cup \Theta)$ and the premises do not interfere.

Given a set $\mathcal{T}$ of transitions, called *axioms*, a $\mathcal{T}$*-computation*, or just *computation* for short, is a sequence of transitions $G_0 \xrightarrow{\Lambda_1} G_1 \xrightarrow{\Lambda_2} \dots$ each of which is derived from the axioms in $\mathcal{T}$ by zero or more applications of the rule [sync].

Example.

Non-interference ensures that no "accidental" attachment of hyperedges ever occurs as a result of applying [sync]. Intuitively, the condition forbids the new nodes appearing to the right hand side of a transition, but not to the left, to be used as links for connecting subgraphs, except if by explicit synchronisation. Violating non-interference, for example, we could put two transitions $P(x) \xrightarrow{\emptyset} R(x, z)$ and $Q(x) \xrightarrow{\emptyset} S(x, z)$ in parallel and deduce the rewrite $P(x)|Q(x) \xrightarrow{\emptyset} R(x, z)|S(x, z)$, where S and T are attached by z with no synchronisation occurring. If such a rewrite was meant, it could be obtained by synchronising the non interfering transitions $P(x) \xrightarrow{(x,a,z)} R(x, z)$ and $Q(x) \xrightarrow{(x,\overline{a},y)} S(x, y)$, which yields a rewrite $P(x)|Q(x) \xrightarrow{\Lambda} R(x, z)|S(x, z)$, where $\Lambda = \{(x, a, z), (x, \overline{a}, y)\}$.

Productions.

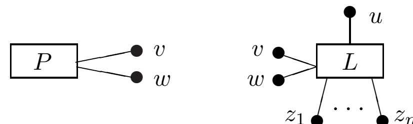
It is often convenient to define the set $\mathcal{T}$ of SHR axioms of a specific computational theory by means of a set of axiom schemes called *productions*. More precisely, a production is a pre-transition with *metavariables*, place holders ranging over $\mathcal{N}$, in the place of nodes. In accordance with [5], we require that the left hand side of a production consist of a single edge. We use the same symbols x, y, z... for nodes and for their place holders: the context will clarify what is meant.

A *faithful instance* of a production is a pre-transition obtained by instanciating all metavariables with specific nodes in $\mathcal{N}$, and such that, if two distinct metavariables are instanciated with the same node, then they must both appear in the left side of the production. The axioms *generated* by a set $\mathcal{P}$ of productions are all transitions $L(\boldsymbol{x}) \xrightarrow{\Lambda} H$ such that $L(\boldsymbol{x}) \xrightarrow{\Lambda} G$ is a faithful instance of a production in $\mathcal{P}$ and $H = \rho(G)$, where $\rho = \upsilon(\Lambda)$. Thus shall we specify the axioms of the SHR theory of DCCS and Mobile Ambients.

Ambient Graphs.

In section 3 and section 4 we develop two case studies where SHR, in the version proposed above, is used to model DCCS, the distributed CCS of [7], and the calculus of Mobile Ambients. Both models adopt a common recursive graph architecture whose components we call *ambient graphs*. Interpretation maps DCCS terms to *flat* ambient graphs, while mobile ambients are trees.

Ambient graphs feature two kinds of edges: $L(u, v, w, \boldsymbol{z})$, called *location managers*, and $P(v, w)$ representing processes running at specific locations. We use the metavariables u, v, w and $\boldsymbol{z} = z_1 \ldots z_n$ consistently to denote nodes attached to specific hyperedge tentacles. Hence, u always denotes a node where the first tentacle of a location manager is attached, and so forth. Local processes and location managers are represented graphically by the following icons.



An ambient graph $\mathcal{A}$ is composed of a location manager L, a subgraph $\mathcal{P}$ of local processes, and a subgraph $\mathcal{S}$ of subambients. This is expressed formally by the following grammar:

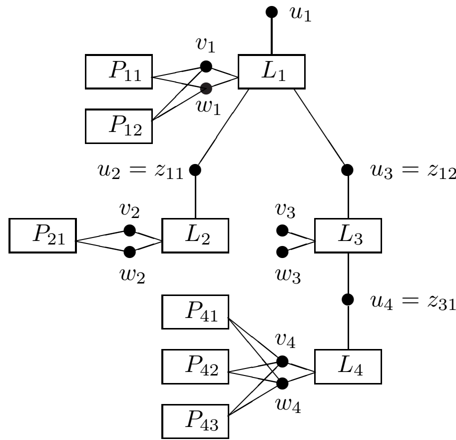


Table 2
A tree-like ambient graph

$$\mathcal{A}(u) ::= L(u, v, w, \boldsymbol{z}) \mid \mathcal{P}(v, w) \mid \mathcal{S}(\boldsymbol{z})$$
$$\mathcal{S}(\boldsymbol{z}) ::= \mathcal{A}_1(z_1) \mid \ldots \mid \mathcal{A}_n(z_n)$$
$$\mathcal{P}(v, w) ::= P_1(v, w) \mid \ldots \mid P_m(v, w).$$

It is assumed that no distinct tentacles of a single edge are ever attached to the same node. A *tree-like* ambient graph is one in which, when two distinct location managers $L(u, v, w, \boldsymbol{z})$ and $L'(u', v', w', \boldsymbol{z}')$ share nodes, it is either by an equation $u = z'_i$ or else by an equation $u' = z_j$. An example of such a structure is given in table 2. More general topologies, which we shall not investigate in the present paper, are obtained by allowing $\mathcal{A}(u)$ to share its node u with more than one location manager. In the language of mobile ambients this would correspond to different ambients sharing common subambients.

An ambient graph $\mathcal{A}(u)$ synchronises with its environment using the node u and with its subambients using the z_i. The nodes v and w connect local processes to the location manager. This allows the latter to act not only as a router for net synchronisation, but also as a go-between in local communication. We found this approach less expensive (in terms of graph complexity) than allowing direct communication of local processes. For example, in the calculus of mobile ambients (see section 4), local communication may unleash new ambients, and this is bound to require the intervention of an agent (viz. the location manager) to "update" the net. Of course, attaching all local processes of an ambient to the same two nodes reduces local parallelism to

$$(\textsc{Act})\ \frac{l \in L}{L \triangleright l[\alpha.p] \xrightarrow{\alpha} L \triangleright l[p]}$$

$$(\textsc{Kill})\ \frac{l \in L}{L \triangleright l[\mathit{kill}\, m.p] \xrightarrow{\tau} L - \{m\} \triangleright l[p]}$$

$$(\textsc{Live})\ \frac{l \in L \qquad m \in L}{L \triangleright l[\mathit{if}\,(m,p,q)] \xrightarrow{\tau} L \triangleright l[p]}$$

$$(\textsc{Dead})\ \frac{l \in L \qquad m \notin L}{L \triangleright l[\mathit{if}\,(m,p,q)] \xrightarrow{\tau} L \triangleright l[q]}$$

$$(\textsc{Spawn})\ \frac{l \in L}{L \triangleright l[\mathit{move}\, k.p] \xrightarrow{\tau} L \triangleright k[p]}$$

$$(\textsc{Str})\ \frac{P \equiv P' \qquad L \triangleright P \xrightarrow{\alpha} L' \triangleright Q \qquad Q \equiv Q'}{L \triangleright P' \xrightarrow{\alpha} L' \triangleright Q'}$$

$$(\textsc{Comm})\ \frac{L \triangleright P \xrightarrow{a} L' \triangleright P' \qquad L \triangleright Q \xrightarrow{\overline{a}} L' \triangleright Q'}{L \triangleright P|Q \xrightarrow{\tau} L \triangleright P'|Q'}$$

$$(\textsc{Par})\ \frac{L \triangleright P \xrightarrow{\alpha} L' \triangleright P'}{L \triangleright P|Q \xrightarrow{\alpha} L' \triangleright P'|Q}$$

Table 3
The semantics of DCCS

interleaving, while different locations may perform independent actions in a single computational step.

3 A Model of Distributed CCS

In the above presentation of hyperedge replacement we got rid of much of the mathematical structure involved in [5], where SHR was proposed as a foundational framework for modelling global computing. Next we provide evidence that the slender SHR we propose still retains the expressive power of the original system: in the present section we present an encoding of the *distributed* CCS (DCCS) of [7] in SHR and prove it to be a bisimulation up to structural equivalence (theorem 1). The same is done for the calculus of Mobile Ambients in the next section.

The semantics of DCCS is given in table 3. DCCS processes run at specific locations, the set of which we denote by *Loc*. For example,

$$l\,[a.p|\mathit{kill}\ k.q] \mid k\,[\overline{a}.r] \tag{1}$$

is a *located* process featuring two locations: l and k. Two *basic* processes, $a.p$ and $\mathit{kill}\ k.q$, run in parallel at location l, and one, $\overline{a}.r$, at k. There is no nesting of locations (that is, a basic process cannot have a located process as subterm). Communication is binary as in CCS and basic processes located at different sites can synchronise just as when residing at the same location. The term (1), for example, can perform a τ transition to $l\,[p|\mathit{kill}\ k.q] \mid k\,[r]$. Processes may *kill* locations. If a process performs a $\mathit{kill}\ k$ action, then k is immediately removed from the set of the locations that are currently alive (*livesets*). The semantics of DCCS is therefore a labelled transition system on *configurations* $L \rhd P$, where L is a liveset and P a located process. For example, in the liveset $\{l, k\}$ the term (1) has a transition

$$\{l, k\} \rhd l\,[a.p|\mathit{kill}\ k.q] \mid k\,[\overline{a}.r] \xrightarrow{\tau} \{l\} \rhd l\,[a.p|q] \mid k\,[\overline{a}.r]$$

Since the rule for prefixed processes allows $L \rhd k\,[a.p] \xrightarrow{a} L \rhd k\,[p]$ only if $k \in L$, a process on a killed location can no longer move. Note that the action of killing is represented by a τ action. Basic processes can also move to different locations ($\mathit{move}\ k.p$) and check the liveset ($\mathit{if}(k, p, q)$ reduces to p if $k \in L$, or otherwise to q). A basic process is called *thread* if it is not of the form $p|q$. When writing a basic process as $p_1 \mid \ldots \mid p_n$ we implicitly assume that all p_i are threads. For simplicity we do not consider summation, restriction and renaming (see [7] for further detail). Hence, a located process is always of the form

$$P = l_1\,[\,p_{11} \mid \ldots \mid p_{1n_1}] \mid \ldots \mid l_m[\,p_{m1} \mid \ldots \mid p_{mn_m}].$$

For simplicity we dismiss terms of the form $l[\,p] \mid l[q]$, which in DCCS is structurally equivalent to $l[\,p \mid q]$. Without this restriction theorem 1 would still hold up to structural equivalence.

Let the *location sort* $\sigma(P)$ of a located process P be the set of locations which appear in it. For example, the sort of $l\,[\mathit{move}\ k\,.\,\emptyset]$ is $\{l, k\}$. Let P be a process $l_1\,[p_1] \mid \ldots \mid l_n[p_n]$, let $L \rhd P$ be a configuration and let $L \cup \sigma(P)$ be the set $\{l_1, \ldots l_n, l_{n+1}, \ldots l_m\}$. When $i > n$ we call l_i *implicit* in $L \rhd P$. We call *dead* a location in $\sigma(P) - L$. The *interpretation* of the configuration $L \rhd P$ is

an ambient graph $\mathcal{A}_{L \triangleright P}$ defined as follows:

$$\begin{aligned}
\mathcal{A}_{L \triangleright P}(u) &= L_\top(u, v, w, \boldsymbol{z}) \mid \mathcal{S}(\boldsymbol{z}) \\
\mathcal{S}(\boldsymbol{z}) &= \mathcal{A}_{l_1[p_1]}(z_1) \mid \ldots \mid \mathcal{A}_{l_n[p_n]}(z_n) \mid \mathcal{A}_{l_{n+1}[]}(z_{n+1}) \mid \ldots \mid \mathcal{A}_{l_m[]}(z_m) \\
\mathcal{A}_{l[p_i]}(u) &= L_{l_i}(u, v, w) \mid \mathcal{P}_{p_i}(v, w) \qquad \text{(if } l \text{ is alive)} \\
\mathcal{A}_{l[p_i]}(u) &= D_{l_i}(u, v, w) \mid \mathcal{P}_{p_i}(v, w) \qquad \text{(if } l \text{ is dead)} \\
\mathcal{P}_{p_i}(v, w) &= P_{p_{i1}}(v, w) \mid \ldots \mid P_{p_{in_i}}(v, w).
\end{aligned}$$

The root of $\mathcal{A}_{L \triangleright P}$ is an edge $L_\top$ which manages net synchronisation. Since in DCCS all basic processes reside at some location, the v and w tentacles of $L_\top$ are dangling. A hyperedge representing a thread p is labelled by P_p. The information on which location is dead and which is alive is encoded in the labels: the manager of a location l is labelled by L_l if $l \in L$ and by D_l otherwise. The dead and the implicit locations of $L \triangleright P$ are represented in $\mathcal{A}_{L \triangleright P}$. This is because in DCCS processes can move to such locations. Hence, the graph $\mathcal{S}(\boldsymbol{z})$ includes as many ambient graphs as there are locations in $L \cup \sigma(P)$. Note that implicit locations do not host processes. Indeed, by the above clauses, $\mathcal{A}_{l[]}(u)$ is a graph including a location manager with no local processes attached.

Table 4 shows the productions for modelling DCCS. We implicitly assume that all transitions generated by *identity* productions $L(\boldsymbol{x}) \xrightarrow{\emptyset} L(\boldsymbol{x})$ are implicitly available. As usual we let α range over the set $A = \{\tau, a, b \ldots \overline{a}, \overline{b} \ldots\}$ of CCS actions. An edge representing a process $p = \alpha.q$ synchronises with its location manager by issuing an action s_α on the node v (or on w, see rule 3). The manager responds with an $\overline{s_\alpha}$. Moreover, within the same transition, the manager can either try to make p communicate locally by issuing an $\overline{s_{\overline{\alpha}}}$ on w (or respectively on v, rule 15), or it can require the intervention of the net manager $L_\top$ by issuing s_α on its u node (rule 8). While synchronising with the location manager (by issuing an $\overline{s_\alpha}$ on the corresponding tentacle z_i), $L_\top$ may either echo α on its u node (rule 1), or it can broadcast s_α through the net for remote communication (rule 2). Note that, be it local or global, successful communication always involves a τ action on the u node of $L_\top$.

Besides the actions for communication we use a set S of *synchronisation* actions for spawning processes, killing locations and checking the liveset:

$$S = \bigcup_{l \in Loc} \{mv_l, kill_l, then_l, else_l\}.$$

As an example we show how processes move across locations. Note that

Root	1. $L_\top(u,v,w,\boldsymbol{z}) \xrightarrow{(z_i,\overline{s_\alpha})(u,\alpha)} L_\top(u,v,w,\boldsymbol{z})$ 2. $L_\top(u,v,w,\boldsymbol{z}) \xrightarrow[]{\substack{(z_i,\bar\beta,\boldsymbol{x})\\(z_j,\beta,\boldsymbol{x})(u,\tau)}} L_\top(u,v,w,\boldsymbol{z})$ $(*)$
Processes	3. $P_{\alpha.p}(v,w) \xrightarrow{(x,s_\alpha)} \mathcal{P}_p(v,w) \quad x\in\{v,w\}$ 4. $P_{\mathit{kill}\ l.p}(v,w) \xrightarrow{(v,\mathit{kill}_l)} \mathcal{P}_p(v,w)$ 5. $P_{\mathit{if}(l,p,q)}(v,w) \xrightarrow{(v,\mathit{then}_l)} \mathcal{P}_p(v,w)$ 6. $P_{\mathit{if}(l,p,q)}(v,w) \xrightarrow{(v,\mathit{else}_l)} \mathcal{P}_q(v,w)$ 7. $P_{\mathit{move}\ l.p}(v,w) \xrightarrow{(v,\mathit{mv}_l,\ x\,y)} \mathcal{P}_p(x,y)$
Locations	8. $L_l(u,v,w) \xrightarrow{(v,\bar\beta,\boldsymbol{x})(u,\beta,\boldsymbol{x})} L_l(u,v,w)$ $(**)$ 9 $L_l(u,v,w) \xrightarrow{(v,\overline{\mathit{kill}_l})(u,s_\tau)} D_l(u,v,w)$ 10. $L_l(u,v,w) \xrightarrow{(u,\overline{\mathit{kill}_l})} D_l(u,v,w)$ 11. $L_l(u,v,w) \xrightarrow{(v,\overline{\mathit{then}_l})(u,s_\tau)} L_l(u,v,w)$ 12. $L_l(u,v,w) \xrightarrow{(u,\overline{\mathit{then}_l})} L_l(u,v,w)$ 13. $L_l(u,v,w) \xrightarrow{\substack{(u,s_\tau)\\(v,\overline{\mathit{mv}_l},\ v\,w)}} L_l(u,v,w)$ 14. $L_l(u,v,w) \xrightarrow{(u,\overline{\mathit{mv}_l},\ v\,w)} L_l(u,v,w)$ 15. $L_l(u,v,w) \xrightarrow{\substack{(u,s_\tau)\\(v,\overline{s_a})(w,\overline{s_{\bar a}})}} L_l(u,v,w)$
Dead	16. $D_l(u,v,w) \xrightarrow{(u,\overline{\mathit{mv}_l},\ v\,w)} D_l(u,v,w)$ 17. $D_l(u,v,w) \xrightarrow{(u,\bar\beta)} D_l(u,v,w)$ $(***)$
$(*)$ $(**)$ $(***)$	$\beta = s_\alpha$ or $\beta\in S$ $\beta = s_\alpha$ or $\beta=\beta_k\in S$ and $k\neq l$ $\beta\in\{\mathit{kill}_l,\mathit{else}_l\}$

Table 4
Productions for DCCS

in DCCS it is possible to move to a dead location, or to kill one. Hence we must allow the edges representing such locations to respond to *move* and *kill* messages. Besides that, they must answer the question whether they are alive (the *else* action of rule 17). Note that a process is allowed to move to, as well as to kill, the same location where it resides (rules 13 and 9 respectively).

Example.

We derive the SHR transition simulating the following DCCS transition: $\{l,k\} \triangleright k[]|l[move\,k.a] \xrightarrow{\tau} \{l,k\} \triangleright k[a]|l[]$, showing how basic processes move across locations. Left and right hand side of the DCCS transition are interpreted respectively as:

$$\mathcal{A}_1 = L_\top(u,v,w,z_1,z_2)|L_k(z_1,v_1,w_1)|L_l(z_2,v_2,w_2)|P_{move\,k.a}(v_2,w_2) \text{ and}$$
$$\mathcal{A}_2 = L_\top(u,v,w,z_1,z_2)|L_k(z_1,v_1,w_1)|L_l(z_2,v_2,w_2)|P_a(v_1,w_1).$$

To rewrite the former to the latter we first synchronise the root and the location manager of k:

$$\frac{L_\top(u,v,w,z_1,z_2) \xrightarrow{\Lambda} L_\top(u,v,w,z_1,z_2) \qquad L_k(z_1,v_1,w_1) \xrightarrow{\Theta} L_k(z_1,v_1,w_1)}{L_\top(u,v,w,z_1,z_2)|L_k(z_1,v_1,w_1) \xrightarrow{\Lambda\cup\Theta} L_\top(u,v,w,z_1,z_2)|L_k(z_1,v_1,w_1)}$$

where $\Lambda = \{(z_1, mv_k,\, x\,y), (z_2, \overline{mv_k},\, x\,y), (u,\tau)\}$ and $\Theta = \{(z_1, \overline{mv_k},\, v_1\,w_1)\}$. The two vectors $x\,y$ and v_1w_1 are unified because two complementary actions are issued on z_1. So: $x \mapsto v_1$ and $y \mapsto w_1$. Next we synchronise the manager of l and the migrating process:

$$\frac{L_l(z_2,v_2,w_2) \xrightarrow{\Lambda'} L_l(z_2,v_2,w_2) \qquad P_{move\,k.a}(v_2,w_2) \xrightarrow{\Theta'} P_a(v',w')}{L_l(z_2,v_2,w_2)|P_{move\,k.a}(v_2,w_2) \xrightarrow{\Lambda'\cup\Theta'} L_l(z_2,v_2,w_2)|P_a(v',w')}$$

where $\Lambda' = \{(z_2, mv_k,\, x'\,y'), (v_2, \overline{mv_k},\, x'\,y')\}$ and $\Theta' = \{(v_2, mv_k,\, v'\,w')\}$ and $x' \mapsto v'$ and $y' \mapsto w'$. The *non interference* conditions assures that v' and w' are new nodes. A last application of rule [sync] to the two transitions just deduced gives us the transition:

$$\mathcal{A}_1 \xrightarrow{\Lambda\cup\Theta\cup\Lambda'\cup\Theta'} \mathcal{A}_2$$

In the last inference we chose a unifier mapping $v' \mapsto v_1$ and $w' \mapsto w_1$. The only effect of making a different choice would be to produce a graph identical

to $\mathcal{A}_2$ up to a renaming of nodes. The only *observable* action (see section 2) in the above transition is a τ action at u node of the root. $\square$

We write $\mathcal{A}_1 \equiv \mathcal{A}_2$ if $\mathcal{A}_1$ and $\mathcal{A}_2$ are identical up to a renaming of nodes. We write $P \equiv Q$ if P and Q are structurally equivalent DCCS processes. Let $\mathcal{A}(u)$ be an ambient graph; we write $\mathcal{A}(u) \xrightarrow{\alpha} \mathcal{A}'$ if $\mathcal{A}(u) \xrightarrow{\Lambda} \mathcal{A}'$ and the only action observable in Λ is α on u. The following theorem is proven in [8].

Theorem 1 *If $L \rhd P \xrightarrow{\alpha} M \rhd Q$ then $\mathcal{A}_{L \rhd P} \xrightarrow{\alpha} \mathcal{A}'$, where $\mathcal{A}' \equiv \mathcal{A}_{M \rhd Q'}$ and $Q \equiv Q'$. If $\mathcal{A}_{L \rhd P} \xrightarrow{\alpha} \mathcal{A}'$ then $L \rhd P \xrightarrow{\alpha} M \rhd Q$ where $\mathcal{A}' \equiv \mathcal{A}_{M \rhd Q'}$ and $Q \equiv Q'$.*

4 A Model of Mobile Ambients

In this section we use ambient graphs to represent terms of the calculus of *Mobile Ambients* [1]. For simplicity we use a version of this calculus without name restriction. By convention we write P_γ for a term $P_\gamma \equiv P_1 | \cdots P_n$ with P_i of the form $M.Q$ or $!Q$. By P_α we denote a term of the form $n[P_1] | \cdots m[P_k]$. We can use structural equivalence to write any term P as $P_\gamma | Q_\alpha$. A process $n[P_\gamma | Q_\alpha]$ is represented by an ambient graph that has a location manager, a subgraph $\mathcal{P}$ that is the representation of P_γ and a subgraph $\mathcal{S}$ of subambients that is the representation of Q_α. We define a function $[\![\cdot]\!]$, mapping terms to ambient graphs, by means of three auxiliary functions $[\![\cdot]\!]^\pi$, $[\![\cdot]\!]^\alpha$, and $[\![\cdot]\!]^\gamma$.

- $\dfrac{[\![P_\gamma]\!]^\gamma = \mathcal{P}(v,w) \qquad [\![Q_\alpha]\!]^\alpha = \mathcal{S}(\boldsymbol{z})}{[\![P_\gamma | Q_\alpha]\!]^\pi = \mathcal{P}(v,w) | \mathcal{S}(\boldsymbol{z})}$ (with $\{v,w\} \cap \boldsymbol{z} = \emptyset$)
- $\dfrac{[\![P]\!]^\alpha = \mathcal{S}_1(\boldsymbol{x}) \qquad [\![Q]\!]^\alpha = \mathcal{S}_2(\boldsymbol{y})}{[\![P|Q]\!]^\alpha = \mathcal{S}_1(\boldsymbol{x}) | \mathcal{S}_2(\boldsymbol{y})}$ (with $\boldsymbol{x} \cap \boldsymbol{y} = \emptyset$)
- $\dfrac{[\![P]\!]^\pi = \mathcal{P}(v,w) | \mathcal{S}(\boldsymbol{z})}{[\![n[P]]\!]^\alpha = L_n(u,v,w,\boldsymbol{z})) | \mathcal{P}(v,w) | \mathcal{S}(\boldsymbol{z})}$ (with u fresh)
- $[\![0]\!]^\alpha = [\![0]\!]^\gamma = nil$
- $[\![M.P]\!]^\gamma = L_{M.P}(v,w)$
- $\dfrac{[\![P]\!]^\gamma = \mathcal{P}_1(v,w) \qquad [\![Q]\!]^\gamma = \mathcal{P}_2(v,w)}{[\![P|Q]\!]^\gamma = \mathcal{P}_1(v,w) | \mathcal{P}_2(v,w)}$

The ambient graph representing a process $P \equiv P_\gamma | Q_\alpha$ is obtained by adding an artificial topmost ambient $\top$. Hence: $[\![P]\!] = [\![\top[P]]\!]^\alpha$. We say that an ambient graph G *silently rewrites* to G' ($G \xrightarrow{\tau} G'$) if every node that is labelled with an action a is also labelled with the corresponding coaction $\overline{a}$ except the

u-node of the root, $u_\top$, which must be labelled with τ. More formaly $G \xrightarrow{\tau} G'$ if $G \xrightarrow{\Lambda} G'$ with $\Lambda(u_\top) = (\tau, \langle\rangle)$ and, for all $x \neq u_\top$, either $\Lambda(x) = \emptyset$ or $\Lambda(x) = \{(a, \mathbf{y}), (\overline{a}, \mathbf{y})\}$.

The following productions implement rewritings triggered by capabilities. As with DCCS, we assume all identity productions. The metavariable M in the second scheme stands for any *path* (sequence of capabilities).

- $L_{\alpha.P}(v, w) \xrightarrow{(v,\,\alpha,\,\boldsymbol{z})} \mathcal{P}(v, w)|\mathcal{S}(\boldsymbol{z})$ with $\alpha \neq (x)$ and $[\![P]\!]^\pi = \mathcal{P}(v, w)|\mathcal{S}(\boldsymbol{z})$
- $L_{(x).P}(v, w) \xrightarrow{(w,\,(M),\,\boldsymbol{z})} \mathcal{P}(v, w)|\mathcal{S}(\boldsymbol{z})$ with $[\![P[M/x]]\!]^\pi = \mathcal{P}(v, w)|\mathcal{S}(\boldsymbol{z})$
- $L_{!P}(v, w) \xrightarrow{(v,\,rep,\,\boldsymbol{z})} L_{!P}(v, w)|\mathcal{P}(v, w)|\mathcal{S}(\boldsymbol{z})$ with $[\![P]\!]^\pi = \mathcal{P}(v, w)|\mathcal{S}(\boldsymbol{z})$

We now give the productions for edges labelled with ambient names. These productions are grouped by the ambient calculus action they implement. In the following productions an ambient m issues an action called *sinc* to warn its father that something has happend inside m. This action is propagated until it reaches the root $\top$. Since $\top$ can acknoweledge only one such action at a time we are sure that only one ambient action can be done during a transition. We give a full explanation of the production implementing *in*. The others are similar and are just listed in table 5. Consider the following transition:

$$m[in\, n.P|Q]|n[R] \rightarrow n[m[P|Q]|R].$$

Our representation does not allow sibling ambients to communicate directly because they don't share nodes. Their communication musts be mediated by their parent, so four edges are involved in this kind of action: an edge labelled with $in\, n.P$ tells its parent ambient m to enter n; m says to its parent that it wishes to enter n and finally n tells his father that it allows m to enter.

$$L(u, v, w, \boldsymbol{z}) \xrightarrow{\substack{(z_j,\,\overline{n\, accept},\,y)\\ (z_i,\,\overline{enter\, n},\,y)\\ (u,\,sinc,\,\langle\rangle)}} L(u, v, w, \boldsymbol{z_1}) \quad \boldsymbol{z_1} = [z_1, \ldots, z_{1-1}, z_{1+1}, \ldots, z_k]$$

This transition is performed by the parent ambient. Node z_i is shared with the entering ambient m , while node z_j is the node shared with ambient n. Both nodes are labelled with the same fresh node y. When the actions on z_i and z_j are matched by their co-actions node y is unified with the nodes passed by m and n. Hence, these are unified as well. The node z_i does not appear in $\boldsymbol{z_1}$, because m is no longer a child of this edge.

$$L_m(u,v,w,\boldsymbol{z}) \xrightarrow[]{\substack{(v,\,\overline{in\,n},\,\boldsymbol{y})\\(u,\,enter\,n,\,u)}} L_m(u,v,w,\boldsymbol{z},\boldsymbol{y})$$

A vector of new nodes $\boldsymbol{y}$ is created for the new ambients introduced by the reduction of $in\,n.P$. Node u is passed to m's father so that it is unified with the new nodes created by n. It is by this unification that the edge moves to n.

$$L_n(u,v,w,\boldsymbol{z}) \xrightarrow{(u,\,n\,accept,\,y)} L_n(u,v,w,\boldsymbol{z},y)$$

This is the accepting ambient. A new node y is created and passed to n's father so that it will be unified with the one passed by the entering ambient. Ambient n gains a tentacle attached to y becoming in this way m's father.

Example.

We derive the SHR transition corresponding to the following reduction of Mobile Ambients: $n[\,]|m[in\,n.o[\,]] \to n[m[o[\,]]]$.

Let $P = n[\,]|m[in\,n.o[\,]]$ and let $Q = n[m[o[\,]]]$. We derive a transition of the graph $[\![P]\!] = L_\top(u,v,w,z_1,z_2)|L_n(z_2,v_2,w_2)|L_m(z_1,v_1,w_1)|L_{in\,n.o[]}(v_1,w_1)$ to $[\![Q]\!] = L_\top(u,v,w,z_2)|L_n(z_2,v_2,w_2,z_1)|L_m(z_1,v_1,w_1,z_3)|L_o(z_3,v_3,w_3)$. We first synchronise the root with the manager of n:

$$\frac{L_\top(u,v,w,z_1,z_2) \xrightarrow{\Lambda} L_\top(u,v,w,z_1) \qquad L_n(z_2,v_2,w_2) \xrightarrow{\Theta} L_n(z_2,v_2,w_2,x)}{L_\top(u,v,w,z_1,z_2)|L_n(z_2,v_2,w_2) \xrightarrow{\Lambda\cup\Theta} L_\top(u,v,w,z_2)|L_n(z_2,v_2,w_2,x)}$$

where $\Lambda = \left\{\begin{smallmatrix}(z_2,\,\overline{n\,accept},\,y)\\(z_1,\,enter\,n,\,y)\\(u,\,\tau,\,\langle\rangle)\end{smallmatrix}\right\}$, $\Theta = \left\{\ {\scriptstyle (z_2,\,n\,accept,\,x)}\ \right\}$.

Notice that nodes x and y must be unified because two actions are defined on z_2. Without loss of generality, we choose the unifier that maps y to x (see discussion in section 3). Now we apply $[sync]$ to the entering ambient and to the process

$$\frac{L_m(z_1,v_1,w_1) \xrightarrow{\Lambda'} L_m(z_1,v_1,w_1,z_3) \qquad L_{in\,n.o[]}(v_1,w_1) \xrightarrow{\Theta'} L_o(z,v_3,w_3)}{L_m(z_1,v_1,w_1|L_{in\,n.o[]}(v_1,w_1) \xrightarrow{\Lambda'\cup\Theta'} L_m(z_1,v_1,w_1,z_3)|L_o(z_3,v_3,w_3)}$$

Where $\Lambda' = \left\{\begin{smallmatrix}(v_1,\,\overline{in\,n},\,z_3)\\(z_1,\,enter\,n,\,z_1)\end{smallmatrix}\right\}$, $\Theta' = \left\{{\scriptstyle (v_1,\,in\,n,\,z)}\ \right\}$. Here nodes z, z_3 must be unified because of the actions defined on v_1. We also underline that the *non interference* conditions assures that v_3, w_3 are new nodes. Using the two transitions just derived as premises, a last application of [sync] yields the transition $[\![P]\!] \xrightarrow{\Lambda\cup\Theta\cup\Lambda'\cup\Theta'} [\![Q]\!]$. Note that node x to the right has been replaced by z_1.

□

out n	$L(u,v,w,\boldsymbol{z_1} z\, \boldsymbol{z_2}) \xrightarrow{\substack{(z,\,\overline{exit\,m},\,y)\\(u,\,\overline{accept},\,y)}} L(u,v,w,\boldsymbol{z_1 z_2})$ $L(u,v,w,\boldsymbol{z}) \xrightarrow{\substack{(v,\,\overline{out\,n},\,\boldsymbol{y})\\(u,\,exit\,n,\,u)}} L(u,v,w,\boldsymbol{z},\boldsymbol{y})$ $L(u,v,w,\boldsymbol{z}) \xrightarrow{\substack{(z_i,\,accept,\,y)\\(u,\,sinc,\,\langle\rangle)}} L(u,v,w,\boldsymbol{z},y)$
open n	$L(u,v,w,\boldsymbol{z_1} z\, \boldsymbol{z_2}) \xrightarrow{\substack{(v,\,\overline{open\,n},\,\boldsymbol{y})\\(z,\,open\,n,\,v\,w\,\boldsymbol{x})\\(u,\,sinc,\,\langle\rangle)}} L(u,v,w,\boldsymbol{z_1 z_2},\boldsymbol{y},\boldsymbol{x})$ $L(u,v,w,\boldsymbol{z}) \xrightarrow{(u,\,\overline{open\,n},\,v\,w\,\boldsymbol{z})} nil$
communication and replication	$L(u,v,w,\boldsymbol{z}) \xrightarrow{\substack{(z_i,\,\overline{sinc},\,\langle\rangle)\\(u,\,sinc,\,\langle\rangle)}} L(u,v,w,\boldsymbol{z})$ $L(u,v,w,\boldsymbol{z}) \xrightarrow{\substack{(u,\,sinc,\,\langle\rangle)\\(v,\,\overline{(M)},\,\boldsymbol{x})\\(w,\,\overline{\langle M\rangle},\,\boldsymbol{y})}} L(u,v,w,\boldsymbol{z},\boldsymbol{x},\boldsymbol{y})$ $L(u,v,w,\boldsymbol{z}) \xrightarrow{\substack{(u,\,sinc,\,\langle\rangle)\\(v,\,\overline{rep},\,\boldsymbol{x})}} L(u,v,w,\boldsymbol{z},\boldsymbol{x})$

Table 5
SHR Productions for Mobile Ambients

Table 5 shows the production for the *open* and *out* capabilities. Besides these there are productions for propagating a *sinc* action, for controlling communication between local processes and for letting a process replicate itself. The corresponding productions of the root location manager $\top$ are omitted because they are identical to the others except that its u node is labelled with τ rather than with *sinc*.

The following theorem is proven in [8].

Theorem 2 *Let P be a process; if $P \to Q$ then $[\![P]\!] \xrightarrow{\tau^*} [\![Q']\!]$ where $Q \equiv Q'$. Let $G = [\![P]\!]$ be a graph; if $G \xrightarrow{\tau} G'$ then either $G' = [\![P']\!]$ and $P \equiv P'$ or $P \to Q$ and $G' = [\![Q]\!]$.*

5 The Synchronised Hypergraph Environment

SHR is the intermediate language of the *Synchronised Hypergraph Environment* (*She*), a tool for system design and software development supporting concurrent and distributed programming. The system, whose architecture is sketched in figure 1, is currently under development, but a prototype is available at `http://briantb.unixcab.org/she/`. *She* is multilingual: the user can chose among different languages for which different editors are provided. Programs are translated into SHR specifications, that is hypergraphs and productions to be applied for rewriting graphs. Besides SHR itself, which the

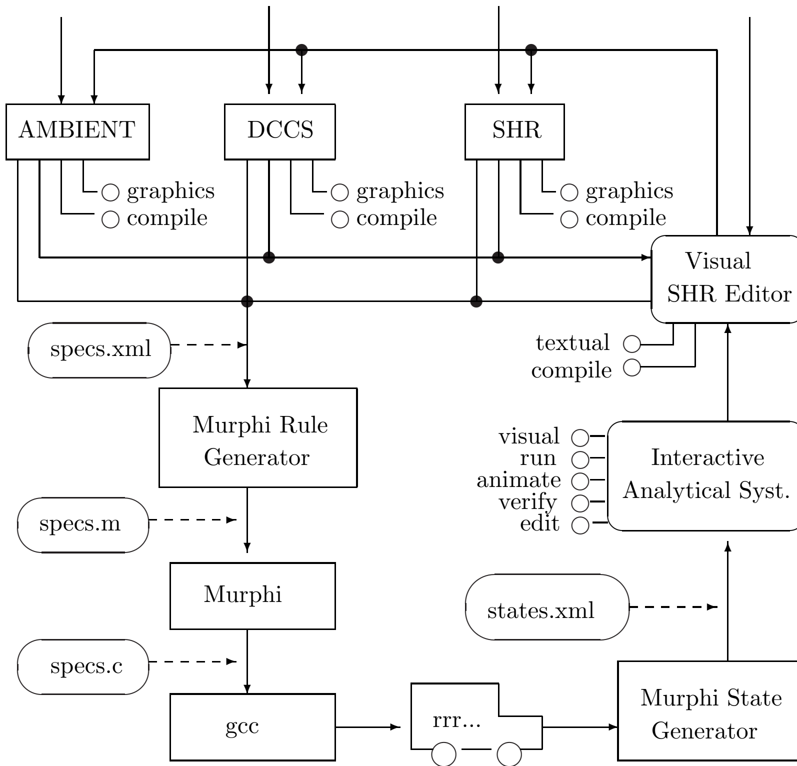


Figure 1. She

user can choose for programming in a declarative style, the system currently supports DCCS and the calculus of Mobile Ambients, whose translations are described in sections 3 and 4. Note, however, that the system is open to the addition of new language modules.

Programs are compiled into the model checker *Murphi* [3]. Murphi is, in fact, both a state verifier and a language: The language is used to specify an initial state (viz. the representation of a hypergraph) and rules for state transition (representing the hypergraph rewritings). A set of conditions, called *invariants* may also be provided. The state verifier computes every possible transition and checks that all invariants hold at all reachable states. The *State Generator* module turns Murphi's output into a graph representing the state space, which can then be explored by the *Interactive Analytical System*. Here the user is provided with visual tools. Specific execution traces can be selected and the hypergraphs representing each state can be visualised in an animation on the screen. It is also possible to animate the possible runs discovered by Murphi violating the invariants. The hypergraphs representing states gener-

ated by the system may be saved and manipulated by the *Visual SHR Editor* or, in some cases, translated back into the original calculus. Within the editor it is possible to write productions or design system architectures graphically. The visual editor feeds back to the Murphi rule generator thus closing the cycle from implementation to testing and back.

Other systems exist which support concurrent programming. Most notably, the *Concurrency Workbench* [2] performs analysis of Temporal CCS and provides a modal logic to check properties and verify observational equivalence on terms. Similarly, LTSA [6] verifies concurrent systems specified in process algebra notation. LTSA does not perform a complete state exploration but can perform an interactive simulation, both in textual and visual fascion. It can also check some simple predefined properties and, if a violation is found, it displays the shortest trace of moves. The novelty that *She* brings with respect to these and similar systems is its extendibility to different language frameworks, due to the simplicity and expressiveness of its intermediate language, SHR. The declarative nature of SHR allows an incremental approach to system design while graphical representation of computational states allows *She* to give visual account of complex network interaction occurring in global computing.

References

[1] L. Cardelli and A.D. Gordon. Mobile Ambients. *Theoretical Computer Science*, 1(240):177–213, 2000.

[2] R. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench. *Springer LNCS*, 407:24–37, 1989.

[3] Alan J. Hu David L. Dill, Andreas J. Drexler and C. Han Yang. Protocol verification as a hardware design aid. In *IEEE Int.Conf. on Computer Design: VLSI in Computers and Processors*, pages 522–525, 1992.

[4] P. Degano and U. Montanari. A model of distributrd systems based on graph rewriting. *Journal of the ACM*, 34:411–449, 1987.

[5] G. Ferrari, U. Montanari, and E. Tuosto. A LTS semantics of ambients via graph synchronization with mobility. *ITCS*, 2001.

[6] Labelled Transition System Analysis. http://www-dse.doc.ic.ac.uk/ concurrency/ ltsa/ LTSA.html.

[7] J. Riely and M. Hennessy. Distributed Processes and Location Failures. *Theoretical Computer Science*, 266:693–735, 2001.

[8] A. Tiberi. Distributed computation as hypergraph rewriting. Masters thesis - forthcoming, 2004.