Electronic Communications of the EASST
Volume X (2010)

International Colloquium on Graph and Model Transformation – On the occasion of the 65th birthday of Hartmut Ehrig (GMT 2010)

Modeling a Service and Session Calculus with Hierarchical Graph Transformation

Roberto Bruni, Andrea Corradini, and Ugo Montanari

15 pages

Guest Editors: Claudia Ermel, Hartmut Ehrig, Fernando Orejas, Gabriele Taentzer
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Modeling a Service and Session Calculus with Hierarchical Graph Transformation*

Roberto Bruni, Andrea Corradini, and Ugo Montanari

[bruni,andrea,ugo]@di.unipi.it
Dipartimento di Informatica, Università di Pisa, Italy

Abstract: Graph transformation techniques have been applied successfully to the modelling of process calculi, for example for equipping them with a truly concurrent semantics. Recently, there has been an increasing interest towards hierarchical structures both at the level of graph-based models, in order to represent explicitly the interplay between linking and containment (like in Milner's bigraphs), and at the level of process calculi, in order to deal with several logical notions of scoping (ambients, sessions and transactions, among others). In this paper we show how to encode a sophisticated calculus of services and nested sessions by exploiting a suitable flavour of hierarchical graphs. For the encoding of the processes of this calculus we benefit from a recently proposed algebra of graphs with nesting.

Keywords: Hierarchical graphs, service oriented architecture, process calculi, CaSPiS

1 Introduction

The use of graphs or diagrams of various kinds is pervasive in Computer Science, as they are very handy for describing in a two-dimensional space the logical or topological structure of systems, models, states, behaviors, computations, metamodels, and several other entities of interest; well-known examples are the graphical presentations of data structures (like lists and trees), of entity-relationship diagrams, of various kinds of automata and labeled transition systems, of static and behavioral UML diagrams (like class, message sequence and state diagrams), of computational formalisms like Petri nets, and so on.

The advantage of drawing graphs or diagrams, rather than using their underlying set-theoretical definition or some term-like linear syntax, lies in the fact that graphs emphasize relevant topological features of the systems or models they describe, like adjacency and connectivity of components, sharing of data and structures, causal dependencies, hierarchical structuring, among others, making such features easily understandable and detectable also to non-specialists. In several cases graphs provide a representation of models or systems at the "right" level of abstraction: for example, as drawings are always understood "up to isomorphism", the order in which nodes and arcs are drawn is typically irrelevant (unless some tacit drawing convention is enforced) and if the concrete identity of certain entities is irrelevant (e.g., the names of the states of a finite state automata), it is sufficient to omit them in the drawing.

The use of graphs as a domain for the visualization of algebraically-specified systems, in general, and process calculi, in particular, has been pursued in a vast literature of which it is not possible

* Research supported by the EU FP6-IST IP 16004 SENSORIA.

to give a comprehensive account here (see, e.g., [BL05] and references therein), but one striking example is the research on "optimal" implementations for functional calculi [AG98]. Here we restrict the attention to the analysis of the concurrent behavior of process calculi with name passing, in the style of [MP95, Gad03, BMM06]. To this aim, there are several "graphical specification frameworks" which provide general techniques and/or tools for the graphical description of systems and, possibly, of their behavior, including Graph Transformation [Roz97], Bigraphical Reactive Systems [Mil06] and Synchronized Hyperedge Replacement [FHL+06].

Recently, there has been an increasing interest towards hierarchical structures both at the level of graph-based models, in order to represent explicitly the interplay between linking and containment (like in Milner's bigraphs), and at the level of process calculi, in order to deal with several logical notions of scoping (ambients, sessions, and transactions, among others). The goal of the work summarized in this paper is to show how to encode both the static aspects and the dynamics of CaSPiS, a sophisticated calculus of services and nested sessions [BBDL08], by exploiting a suitable flavor of hierarchical graphs and corresponding transformation rules.

Following a methodological approach that has been applied recently to provide a graphical encoding of the static aspects of a variety of formalisms (including process calculi, workflow languages, entity relationship diagrams and others, see [BGL10c, BGL10a]), we will not present the graph encoding of CaSPiS processes directly, but we will exploit instead as an intermediate language a recently proposed algebra of hierarchical designs, which allows to reduce the representation distance between the considered formalisms.

The algebra is defined by an equational signature, whose operator symbols are interpreted as operations on graphs, and where the axioms formalize suitable properties of such operators. Therefore the terms of the initial algebra can be interpreted as graphs, and the axioms can be shown to be sound and complete with respect to the interpretation, in the sense that two terms are equivalent if and only if they denote the same graph (up to isomorphism). The interesting fact, is that the interpretation of the terms of the algebra can be given over different kinds of graphs, resulting in different layouts. As a typical example, the nested structure of designs can be interpreted adequately in a class of truly hierarchical graphs, where subgraphs can be encapsulated in hyperedges, or also can be rendered by over-imposing a tree representing the hierarchy to a standard, flat hypergraph, .

Therefore, the advantages of the use of an intermediate algebra for the encoding are twofold:

- the algebra provides explicit operators for parallel composition, nesting of components, names representing shared resources, local and global restriction, as well as aliasing mechanisms: the richness of such operators makes the encoding of process algebras like CaSPiS quite intuitive, less error-prone and easy to understand;
- the various interpretations of the terms of the algebra as different kinds of graphs can be defined once and for all, and reused for the encoding of several other formalisms.

In the next sections we shall first account for the algebra of hierarchical graphs, sketching, only at the informal level, how it can be interpreted over both hierarchical graphs and term graphs. Next we introduce the syntax and the reduction semantics of (a significant fragment of) CaSPiS, and show how the static aspects of CaSPiS can be encoded in the algebra. As far as the dynamics of CaSPiS processes is concerned, the work is still ongoing, but some interesting aspects will be

discussed. In particular, as the CaSPiS reduction semantics allows for reactions in (static) contexts of arbitrary depth, the standard notion of graph transformation rule, which has a local effect only, is not sufficient to model it. We will sketch some possible approaches to overcome this problem.

Some preliminary work on the graphical encoding of CaSPiS and its behavioral semantics has been presented in [Ter08].

2 An algebra of hierarchical graphs

We introduce here our algebra of (typed) hierarchical graphs that we call *designs*. The algebraic presentation of designs is inspired by our previous work on *Architectural Design Rewriting* [BLMT08] (hence the name) and by the graph algebra of CHARM [CMR94].

Definition 1 (design) A *design* is a term of sort $\mathbb{D}$ generated by the grammar

$$\mathbb{D} \;::=\; L_{\bar{x}}[\mathbb{G}] \qquad \mathbb{G} \;::=\; \mathbf{0} \mid x \mid l\langle\bar{x}\rangle \mid \mathbb{G} \mid \mathbb{G} \mid (\nu x)\mathbb{G} \mid \mathbb{D}\langle\bar{x}\rangle$$

where l and L are drawn from disjoint vocabularies $\mathscr{E}$ and $\mathscr{D}$ of *edge* and *design labels*, respectively, x is taken from a global set $\mathscr{N}$ of *nodes*, and $\bar{x} \in \mathscr{N}^*$ is a list of nodes.

As a matter of notation, in the following, we let $\lfloor\bar{x}\rfloor$ denote the set of elements of a list $\bar{x}$ and overload $|\cdot|$ to denote both the length of a list and the cardinality of a set.

Terms generated by $\mathbb{G}$ and $\mathbb{D}$ are meant to represent hierarchical graphs and "edge-encapsulated" hierarchical graphs, respectively. The syntax has the following informal meaning: $\mathbf{0}$ represents the empty graph, x is a discrete graph containing node x only, $l\langle\bar{x}\rangle$ is a graph formed by an l-labeled (hyper)edge attached to nodes $\bar{x}$ (the i-th tentacle to the i-th node in $\bar{x}$, sometimes denoted by $\bar{x}[i]$), $\mathbb{G} \mid \mathbb{H}$ is the graph resulting from the parallel composition of graphs $\mathbb{G}$ and $\mathbb{H}$ (their disjoint union up to shared nodes), $(\nu x)\mathbb{G}$ is the graph $\mathbb{G}$ after making node x not visible from the outside (borrowing nominal calculus jargon we say that the node x is *restricted*), and $\mathbb{D}\langle\bar{x}\rangle$ is a graph formed by attaching design $\mathbb{D}$ to nodes $\bar{x}$ (the i-th node in the interface of $\mathbb{D}$ to the i-th node in $\bar{x}$).

A term $L_{\bar{x}}[\mathbb{G}]$ is a design labeled by L, with body graph $\mathbb{G}$ whose nodes $\bar{x}$ are exposed in the interface. To clarify the exact role of the interface of a design, we can use a programming metaphor: a design $L_{\bar{x}}[\mathbb{G}]$ is like a procedure declaration where $\bar{x}$ is the list of formal parameters. Then term $L_{\bar{x}}[\mathbb{G}]\langle\bar{y}\rangle$ represents the application of the procedure to the list of actual parameters $\bar{y}$; of course, in this case $\bar{x}$ and $\bar{y}$ must have the same length.

Restriction $(\nu x)\mathbb{G}$ acts as a binder for x in $\mathbb{G}$ and similarly $L_{\bar{x}}[\mathbb{G}]$ binds $\lfloor\bar{x}\rfloor$ in $\mathbb{G}$, leading to the usual notion of *free* nodes $fn(\mathbb{D})$ and $fn(\mathbb{G})$, defined inductively as follows:

$$fn(L_{\bar{x}}[\mathbb{G}]) = fn(\mathbb{G}) \setminus \lfloor\bar{x}\rfloor \qquad fn(\mathbf{0}) = \emptyset \qquad fn(x) = \{x\} \qquad fn(l\langle\bar{x}\rangle) = \lfloor\bar{x}\rfloor$$
$$fn(\mathbb{G} \mid \mathbb{H}) = fn(\mathbb{G}) \cup fn(\mathbb{H}) \qquad fn((\nu x)\mathbb{G}) = fn(\mathbb{G}) \setminus \{x\} \qquad fn(\mathbb{D}\langle\bar{x}\rangle) = fn(\mathbb{D}) \cup \lfloor\bar{x}\rfloor$$

The algebra includes the structural graph axioms of [CMR94] such as associativity and commutativity for $\mid$ with identity $\mathbf{0}$ (axioms DA1–DA3 in Definition 2) and restricted nodes (DA4–DA6). In addition, it includes axioms to α-rename bound nodes (DA7–DA8), an axiom for making immaterial the addition of a node to a graph where that same node is already free (DA9) and another one ensuring that free nodes are not localized within hierarchical edges (DA10).

Definition 2 ($\equiv_D$) The structural congruence $\equiv_D$ over well-formed designs and graphs is the least congruence satisfying the axioms in Fig. 1, where in axiom (DA7) the substitution is required to be a function (to avoid node coalescing).

$$
\begin{array}{rcll}
\mathbb{G} \mid \mathbb{H} & \equiv & \mathbb{H} \mid \mathbb{G} & & \text{(DA1)} \\
\mathbb{G} \mid (\mathbb{H} \mid \mathbb{I}) & \equiv & (\mathbb{G} \mid \mathbb{H}) \mid \mathbb{I} & & \text{(DA2)} \\
\mathbb{G} \mid \mathbf{0} & \equiv & \mathbb{G} & & \text{(DA3)} \\
(\nu x)(\nu y)\mathbb{G} & \equiv & (\nu y)(\nu x)\mathbb{G} & & \text{(DA4)} \\
(\nu x)\mathbf{0} & \equiv & \mathbf{0} & & \text{(DA5)} \\
\mathbb{G} \mid (\nu x)\mathbb{H} & \equiv & (\nu x)(\mathbb{G} \mid \mathbb{H}) & \text{if } x \notin fn(\mathbb{G}) & \text{(DA6)} \\
L_{\bar{x}}[\mathbb{G}] & \equiv & L_{\bar{y}}[\mathbb{G}\{{}^{\bar{y}}/_{\bar{x}}\}] & \text{if } \lfloor \bar{y} \rfloor \cap fn(\mathbb{G}) = \emptyset & \text{(DA7)} \\
(\nu x)\mathbb{G} & \equiv & (\nu y)\mathbb{G}\{{}^{y}/_{x}\} & \text{if } y \notin fn(\mathbb{G}) & \text{(DA8)} \\
x \mid \mathbb{G} & \equiv & \mathbb{G} & \text{if } x \in fn(\mathbb{G}) & \text{(DA9)} \\
L_{\bar{x}}[z \mid \mathbb{G}]\langle \bar{y} \rangle & \equiv & z \mid L_{\bar{x}}[\mathbb{G}]\langle \bar{y} \rangle & \text{if } z \notin \lfloor \bar{x} \rfloor & \text{(DA10)}
\end{array}
$$

Figure 1: Structural congruence axioms for designs

It is immediate to observe that structural congruence respects free nodes, i.e. $\mathbb{G} \equiv_D \mathbb{H}$ implies $fn(\mathbb{G}) = fn(\mathbb{H})$ for any $\mathbb{G}, \mathbb{H}$. Moreover, being $\equiv_D$ a congruence, it is closed w.r.t. all operators; in particular, we have $L_{\bar{x}}[\mathbb{G}] \equiv_D L_{\bar{x}}[\mathbb{H}]$ whenever $\mathbb{G} \equiv_D \mathbb{H}$.

Two different classes of models have been studied for our design algebra, as summarized in the next two subsections: these are in straight analogy with two common visual representations of file systems. The icon view, where each folder is a window recursively containing files and folders, represents a global view of the system taken "from the top". Instead the tree-like view, where the whole hierarchy is presented as a tree whose nodes can be contracted and expanded and where containment is rendered, for example, through indentation, represents some sort of "side-view" of the system.

2.1 Top-view models

In [BGL10b] we have proposed an original notion of hierarchical graphs with interfaces: roughly they extend ordinary hyper-graphs with the possibility to embed (recursively) a hierarchical graph within each edge, thus inducing a layered structure of nodes and edges. Notably, differently from the definition proposed in [DHP02], the nodes defined in one layer are also visible below in the hierarchy (but not above). The main result of [BGL10b] shows that the encoding of design terms in hierarchical graphs is surjective and that the axiomatization of the design algebra is sound and complete w.r.t. the encoding. Moreover, in presence of the extrusion axiom, which is introduced later (see Definition 3), the encoding can be slightly modified in order to preserve the validity of the main results. The set-theoretical presentation of hierarchical graphs is quite heavy and out of the scope of this paper: we refer the interested reader to [BGL10b] for all technical details and formal proofs.

The following example gives a better intuition of the algebra and of the model of hierarchical graphs. For this purpose we use an informal, appealing visual notation.

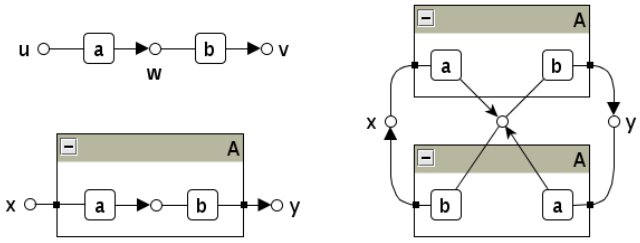

Figure 2: The hierarchical graphs corresponding to some terms of the graph algebra

Example 1 *Let* $a,b \in \mathscr{E}$, $A \in \mathscr{D}$, $u,v,w,x,y \in \mathscr{N}$. *We depict in Fig. 2 the hierarchical graphs corresponding to a few terms of our algebra. Nodes are represented by circles and free nodes are annotated with their name. Edges are represented by rounded boxes, annotated inside with the edge label. Each design is represented by a rectangular box with the label in a top bar, and encapsulating the body graph. Instead of numbering the tentacles of edges and designs, we use different kinds of lines and arrows: in this example the first tentacle of an edge is represented by a plain line, while the second one is denoted by a standard arrow.*

To simplify the drawings, the interface nodes of a design are drawn as small black boxes on its border, and tentacles connected to them are prolonged to the corresponding nodes which the design is attached to.

Figure 2 includes the graphs corresponding to the following terms: $\mathbb{G} = a\langle u,w\rangle \mid b\langle w,v\rangle$ *(top-left),* $A_{u,v}[(\nu w)\mathbb{G}]\langle x,y\rangle$ *(bottom-left), and* $(\nu w)(A_{u,v}[\mathbb{G}]\langle x,y\rangle \mid A_{u,v}[\mathbb{G}]\langle y,x\rangle)$ *(right).*

In the last graph, note the difference among the tentacles connected to x *and* y*, and those pointing to the restricted node in the middle. The formers cross the border of the designs, reaching* x *or* y *through the exposed interface nodes, while the latters access the restricted node directly, as it is available globally.*

The hierarchical graphs in Fig. 2 illustrate a global view as taken from the top. Another possibility is to take a side-view, where containment is represented by dependencies between items in different layers.

2.2 Side-view models

In [BCG+] we have followed the tree-like analogy to define a second interpretation of the design algebra, over a class of graphs called *gs-graphs* [FM00]. Roughly, gs-graphs are an extension of term-graphs [BEG+87] tailored to many-sorted hyper-signatures. In fact, it is known that the term-graphs over a standard one-sorted signature Σ can be generated freely from Σ itself by closing it with respect to the axioms of *gs-monoidal theories* (see [CG99]). The same construction works in the many-sorted case as well, and even if the operators in Σ can deliver an arbitrary, finite number of results, instead of exactly one: in this case the result of the construction is the set of all gs-graphs over Σ.

For the interpretation of the design algebra, we therefore fix a (many-sorted, hyper-) signature

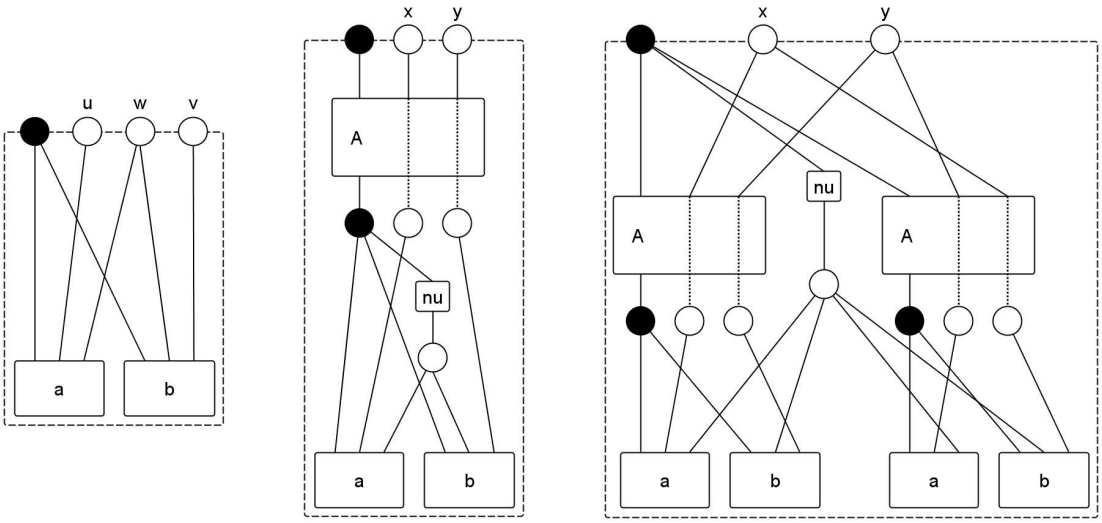

Figure 3: Hierarchical structures as gs-graphs

Σ_D with one sort $\circ$ for nodes and an additional one, denoted $\bullet$, to represent locations. Assuming that the labels in $\mathscr{E} \cup \mathscr{D}$ have fixed ranks (see the next subsection), Σ_D includes an operator $l\colon \bullet\circ^k \to \varepsilon$ for each edge label $l \in \mathscr{E}$ of rank k, where ε denotes the empty list of sorts, and an operator $L\colon \bullet\circ^k \to \bullet\circ^k$ for each design label $L \in \mathscr{D}$. Intuitively, all labels have as arguments one location (where the edge/design is placed) and k nodes (which the edge/design is connected to); furthermore design labels offer as results a new location (the interior of the design) and k nodes (the inner formal parameters), while edge labels do not return anything. Finally, Σ_D contains one operator $\mathtt{nu}\colon \bullet \to \circ$ used to encode the (localized) restriction.

Then, the results in [BCG+] define a sound and complete encoding of design terms in gs-graphs over Σ_D.[1] Again, we skip all technical details and we just sketch in Fig. 3 the gs-graphs corresponding to the hierarchical graphs in Fig. 2: $\mathbb{G} = a\langle u,w\rangle \mid b\langle w,v\rangle$ on the left, $A_{u,v}[(\nu w)\mathbb{G}]\langle x,y\rangle$ in the middle, and $(\nu w)(A_{u,v}[\mathbb{G}]\langle x,y\rangle \mid A_{u,v}[\mathbb{G}]\langle y,x\rangle)$ on the right. Each drawing is decorated with an external dashed line enclosing the gs-graph and emphasizing its boundary, on which the names of the available free nodes are placed; furthermore some dotted lines suggests the correspondence between actual and formal parameters of A-labeled edges. Such decorations are not part of the formal definition and have the only purpose of making easier the intuitive correspondence with Fig. 2.

2.3 Well-typedness and extrusion

In practice, it is very frequent that one is interested in disciplining the use of edge and design labels so to be attached only to a specific number of nodes (possibly of specific sorts) or to contain

[1] Actually the construction in [BCG+] is carried out for a slightly different algebra of designs/graphs, but it is discussed how to extend the results to the algebra considered here.

graphs of a specific shape. To this aim it is typically the case that: 1) nodes are sorted, in which case their labels take the form $n:s$ for n the *name* and s the *sort* of the node; and 2) each label of $\mathscr{E}$ and $\mathscr{D}$ has a fixed rank, which in the general case is a finite sequence of sorts. When this is the case, we say that a design (or a graph) is *well-typed* if for each sub-term $L_{\bar{x}}[\mathbb{G}]$ we have that the (lists of) sorts of $\bar{x}$ and L coincide, and similarly for sub-terms $\mathbb{D}\langle\bar{x}\rangle$ and $l\langle\bar{x}\rangle$.

In addition to the axioms of Fig. 1, another axiom that has been considered in the literature is the so-called *extrusion* axiom.

Definition 3 (Extrusion axiom) The extrusion axiom is $L_{\bar{y}}[(\nu z)\mathbb{G}]\langle\bar{x}\rangle \equiv (\nu z)L_{\bar{y}}[\mathbb{G}]\langle\bar{x}\rangle$, for any $L \in \mathscr{D}$, where $z \notin \lfloor\bar{x}\rfloor \cup \lfloor\bar{y}\rfloor$.

The presence of the extrusion axiom implies that restriction of nodes is *global*, i.e., orthogonal to nesting: like free nodes (see axiom DA10), also restricted nodes can cross the boundary of a design. Instead its absence implies that restriction is located to a design.[2] Concerning the side-view encodings using gs-graphs sketched in Section 2.2, the extrusion axiom is easily captured by replacing operator $\mathtt{nu}:\bullet\to\circ$ with $\mathtt{nu}:\varepsilon\to\circ$, meaning that restriction does not take a location as argument. Our encoding of CaSPiS requires the presence of the extrusion axiom, and the different side-view encodings of restriction can be grasped by comparing the nu-labeled edges of Figures 3 and 6.

3 A calculus with nested structures and communication: CaSPiS

This section recalls the basics of CaSPiS [BBDL08], a session-centered calculus. We have chosen this calculus since it represents a non-trivial example of the interplay between nesting and linking in presence of nested sessions, pipelines and communication.

While referring the interested readers to [BBDL08] for an exhaustive description of CaSPiS, we remark that we focus here on the "close-free" fragment of the calculus and we present a slightly simplified syntax (without summation and pattern-matching). Both decisions are for the sake of a convenient and clean presentation only, and constitute no limitation on expressiveness.

CaSPiS is based on the following key computing entities: (i) service definitions $s.P$ and invocations $\bar{s}.Q$, whose synchronization establishes (ii) a fresh session name r shared by the two partner session sides $r\triangleright P$ and $r\triangleright Q$, where respective interaction protocols can interact in both directions by executing (iii) intra-session (synchronous) output $\langle u\rangle$ and input $(?x)$ prefixes. Moreover, (iv) session sides can be nested, and (v) a children side can execute an (extra-session) return prefix $\langle u\rangle^{\uparrow}$ for making u available to its parent session side. Finally, (vi) in-side computation can be achieved using the pipeline operator $P > (?x)Q$, which redirects each output $\langle u\rangle$ from P to activate a corresponding new instance $Q\{^{u}/_{x}\}$ of Q. Notably, any such instance will run in parallel with $P > (?x)Q$. Summarizing all the above, each CaSPiS process can be thought of as running in an environment providing him different means of communication: one channel for "standard" input (expecting values from the partner session side), one channel for "standard" output (either directed to the partner session side or to an in-side pipeline) and one channel for returning values one level up (according to the nesting of session sides).

[2] A different approach is taken in [BCG+], where two distinct restriction operators are introduced.

Definition 4 (CaSPiS syntax) Let $\mathscr{S}$ a set of service names, $\mathscr{R}$ be a set of session names, $\mathscr{V} \supseteq \mathscr{S}$ a set of value names (disjoint from $\mathscr{R}$), and $\mathscr{X} \subseteq \mathscr{V}$ a set of value variables. The set $\mathscr{P}$ of CaSPiS processes is the set of all the terms P generated by the grammar below

$$\begin{array}{lcl} P & ::= & \mathbf{0} \mid r \triangleright P \mid P > Q \mid (\nu w)P \mid P \mid P \mid A.P \\ A & ::= & s \mid \bar{s} \mid (?x) \mid \langle u \rangle \mid \langle u \rangle^{\uparrow} \end{array}$$

where $s \in \mathscr{S}$, $r \in \mathscr{R}$, $u \in \mathscr{V}$, $w \in (\mathscr{V} \cup \mathscr{R}) \setminus \mathscr{X}$ and $x \in \mathscr{X}$.

As usual, the restriction operator $(\nu w)P$ binds w in P, and similarly $(?x).P$ binds x in P, leading to straightforward definition of free names $fn(P)$ of a process P.

Albeit the syntax allows for more general forms of pipelines, for simplicity we only consider pipelines of the form $P > (?x)Q$: these match a standard pattern that, for example, is written as $P > x > Q$ in the Orc programming language [KCM06].

We assume that in any process P at most two session sides are present for the same session name and that the binary relation $\prec_P^+$ over session names is irreflexive, where we write $r \prec_P r'$ whenever in P a session side r' appears nested within a session side r, and $\prec_P^+$ denotes the transitive closure of $\prec_P$.

The operational semantics is defined in terms of reduction rules over processes taken up to a suitable structural congruence, that we introduce next.

Definition 5 ($\equiv_C$) The structural congruence for CaSPiS processes is the relation $\equiv_C \subseteq \mathscr{P} \times \mathscr{P}$, closed under process construction, inductively generated by the axioms in Fig. 4.

$$\begin{array}{rcll} P \mid (Q \mid R) & \equiv & (P \mid Q) \mid R & & (\mathsf{CA1}) \\ P \mid Q & \equiv & Q \mid P & & (\mathsf{CA2}) \\ P \mid \mathbf{0} & \equiv & P & & (\mathsf{CA3}) \\ (\nu n)(\nu m)P & \equiv & (\nu m)(\nu n)P & & (\mathsf{CA4}) \\ (\nu n)\mathbf{0} & \equiv & \mathbf{0} & & (\mathsf{CA5}) \\ P \mid (\nu n)Q & \equiv & (\nu n)(P \mid Q) & \text{if } n \notin fn(P) & (\mathsf{CA6}) \\ ((\nu n)Q) > P & \equiv & (\nu n)(Q > P) & \text{if } n \notin fn(P) & (\mathsf{CA7}) \\ r \triangleright (\nu n)P & \equiv & (\nu n) r \triangleright P & \text{if } n \neq r & (\mathsf{CA8}) \\ (\nu n)P & \equiv & (\nu m)(P\{^m/_n\}) & \text{if } m \notin fn(P) & (\mathsf{CA9}) \\ (?x).P & \equiv & (?y).(P\{^y/_x\}) & \text{if } y \notin fn(P) & (\mathsf{CA10}) \end{array}$$

Figure 4: Structural congruence axioms for CaSPiS.

The reduction rules that we will present make use of contexts; a context $C[\cdot]$ is simply a process term in which there is a single occurrence of a process variable X, called the hole of the context. With $C[P]$ we denote the process obtained by filling the hole of the context with the process P (i.e. we substitute X with P). We can easily generalize such definition to n holes: instead of a single process variable X, we will have n process variables $X_1, \ldots, X_n$.

Definition 6 (Static and dynamic operators) The operators $s.[\cdot]$, $\bar{s}.[\cdot]$, $\langle u \rangle.[\cdot]$, $(?x).[\cdot]$, $\langle u \rangle^{\uparrow}.[\cdot]$ and

$P > [\cdot]$ are *dynamic*. The remaining operators ($r \triangleright [\cdot]$, $[\cdot] > P$, $(\nu n)[\cdot]$ and $P|[\cdot]$) are *static*.

Intuitively the dynamic operators, like the prefixes in the π-calculus or in CCS, do not allow a transition to take place in their argument. We can now define the contexts in which the various kinds of action prefixes are ready to be executed.

Definition 7 (Static and "immune" contexts) A context $C[\cdot]$ is *static* if its hole does not occur in the scope of a dynamic operator. A static context is *session-immune* if the hole does not appear in the scope of a session operator $r \triangleright [\cdot]$. A static context is *pipeline-immune* if the hole does not appear in the scope of a pipeline operator $[\cdot] > P$.

Session-immune contexts are guaranteed not to interfere with inputs and returns of the process in their hole, while contexts that are both session- and pipeline-immune are also guaranteed not to interfere with outputs. Note that in the latter case, the hole can only appear under restriction and parallel composition. We are ready now to present the reduction semantics of CaSPiS.

Definition 8 (Reduction rules of CaSPiS) Given two CaSPiS processes P and Q we have $P \Rightarrow Q$ if and only if one of the five cases in Fig. 5 holds, for some static contexts $C[\cdot]$, $C[\cdot,\cdot]$, some static session-immune contexts $S_0[\cdot]$ and $S_1[\cdot]$, some processes P', P'', R and some names r, r', u and x.

$$
\begin{array}{lrcll}
 & P & \equiv & C[\, s.P' \,,\, \overline{s}.R \,] & \\
(1) & Q & \equiv & (\nu r) C[\, s.P' \,|\, r \triangleright P' \,,\, r \triangleright R \,] & (\mathit{ServiceSync}) \\
 & & & \text{with } r \text{ fresh for } P', C[\cdot], R & \\
 & P & \equiv & (\nu r) C[\, r \triangleright (P' \,|\, \langle u \rangle.P'') \,,\, r \triangleright S_0[(?x).R] \,] & \\
(2) & Q & \equiv & (\nu r) C[\, r \triangleright (P' \,|\, P'') \,,\, r \triangleright S_0[R\{^u/_x\}] \,] & (\mathit{SessionSync}) \\
 & & & \text{with } r \text{ not appearing in } C & \\
 & P & \equiv & (\nu r') C[\, r' \triangleright (P' \,|\, r \triangleright S_0[\langle u \rangle^{\uparrow}.P'']) \,,\, r' \triangleright S_1[(?x).R] \,] & \\
(3) & Q & \equiv & (\nu r') C[\, r' \triangleright (P' \,|\, r \triangleright S_0[P'']) \,,\, r' \triangleright S_1[R\{^u/_x\}] \,] & (\mathit{SessionSyncRet}) \\
 & & & \text{with } r' \text{ not appearing in } C & \\
 & P & \equiv & C[\, (P' \,|\, \langle u \rangle.P'') > (?x).R \,] & \\
(4) & Q & \equiv & C[\, R\{^u/_x\} \,|\, ((P' \,|\, P'') > (?x).R) \,] & (\mathit{PipelineSync}) \\
 & P & \equiv & C[\, (P' \,|\, r \triangleright S_0[\langle u \rangle^{\uparrow}.P'']) > (?x).R \,] & \\
(5) & Q & \equiv & C[\, R\{^u/_x\} \,|\, ((P' \,|\, r \triangleright S_0[P'']) > (?x).R) \,] & (\mathit{PipelineSyncRet})
\end{array}
$$

Figure 5: Possible cases for $P \Rightarrow Q$

The first rule models the invocation of a service: there is a definition of service s ($s.P'$) and a request of invocation of such service ($\overline{s}.R$) located somewhere else in the system. Then a new session r is created with the protocols P' and R of the server and of the client respectively. Note

that differently from [BBDL08], here services are persistent: they are not discarded once invoked and thus they can serve other requests.

Rule (*SessionSync*) allows session partners to exchange messages, through a concretion $\langle u \rangle.P''$ and an abstraction $(?x).R$. Technically, the concretion $\langle u \rangle$ can appear in an arbitrary session- and pipeline-immune context within the session operator, but since restrictions can be moved outside the session operator by structural congruence, this is equivalent to require that the concretion is in parallel with an arbitrary process P', as indicated in the rule. Instead the abstraction $(?x)$ can be at an arbitrary depth in the syntax tree, for example in the left-side of a pipeline, but not in a nested session operator: for this reason we use a static session-immune context $S_0[\cdot]$. The next rule (*SessionSyncRet*) can be used for returning a value computed by a nested session side to the session partner. One can view this rule as composed of two steps: first the value computed in session r is passed to the enclosing session side r', then such session side sends the value to its partner.

The pipeline rule (*PipelineSync*) shows that a value computed by the left-hand side $P' | \langle u \rangle.P''$ can trigger a new instance $R\{^u/_x\}$ of the right-hand side $(?x)R$. Finally the rule (*PipelineSyncRet*) describes the situation where a pipe can be activated through a value returned by a nested session side of the process on the left side of the pipeline.

4 Encoding CaSPiS into the algebra of designs

In [BGL10c, BGL10b] we have provided a sound and complete encoding of CaSPiS processes to our algebra of designs, exploiting the fact that reduction rules can then be directly interpreted over and applied to graphs instead of terms. Unfortunately, this way an interleaving semantics is obtained, not a truly concurrent one, because the whole graph is rewritten at each step (no standard notion of "preserved" nodes/edges is available).

Here we pursue a different objective, by establishing an encoding for which ordinary graph rewriting techniques can be used to recover the dynamics. In particular, as rewrites are forbidden under dynamic contexts of CaSPiS, we will expand dynamic operators only by need. This means that for each term P having a dynamic top operator, we introduce a corresponding edge label, sorted according to the free names of P, which are needed as parameters in the rewrite rule that will expand P to the corresponding graph after a reaction. Instead, the static contexts will be encoded in nested designs corresponding to the session and left-pipeline operators, while restriction operators will be encoded directly as restrictions of the algebra of designs.

In the following we assume that a standard total order on names is available, and for a set of names X we denote by $\lceil X \rceil$ the list of names in X ordered accordingly. Moreover, we assume the existence of a canonical set of totally ordered fresh names $\mathscr{C}$ disjoint from $\mathscr{V} \cup \mathscr{R}$, together with a canonical (order preserving) renaming $\sigma_X : X \to \mathscr{C}$ for any $X \subseteq \mathscr{V} \cup \mathscr{R}$ such that whenever $|X| = |Y|$ then $\sigma_X(X) = \sigma_Y(Y)$. We denote by $can(P)$ the term $P\sigma_{fn(P)}$ obtained by renaming the free names of P according to $\sigma_{fn(P)}$, and we write $\underline{P}$ for one chosen standard representative of the equivalence class $[can(P)]_{\equiv_C}$.

Names (of services, sessions, etc.) are encoded as nodes of the algebra, thus we assume that the set of nodes is sorted accordingly, even if we do not make this formal. The set of edge labels is $\{ \underline{A.P} \}$, i.e. it includes all standard representatives for processes of the form $A.P$. The

tentacles of $\underline{A.P}$ are sorted according to $fn(A.P)$. The set of design labels includes $SES_{_}$ for session sides (exposing an anonymous session name $_$), and one standard representative $\underline{x > Q}$ for each static context of the form $[\cdot] > (?x)Q$, exposing $n = |fn(Q) \setminus \{x\}|$ canonical fresh variables $\sigma_{fn(Q)\setminus\{x\}}(fn(Q) \setminus \{x\})$.

As a matter of notation, for a term $L_{\overline{y}}[\mathbb{G}]\langle\overline{x}\rangle$ we use the shorthand $L[\mathbb{G}]\langle\overline{x}\rangle$ if $\lfloor\overline{y}\rfloor \cap fn(\mathbb{G}) = \emptyset$.

Definition 9 (CaSPiS encoding) The interpretation of CaSPiS operators over the design algebra (with extrusion, i.e., with global restriction) is given by

$$\begin{array}{rcl}
[\![\mathbf{0}]\!] & \stackrel{\text{def}}{=} & \mathbf{0} \\
[\![A.P]\!] & \stackrel{\text{def}}{=} & \underline{A.P}\langle\lceil fn(A.P)\rceil\rangle \\
[\![r \triangleright P]\!] & \stackrel{\text{def}}{=} & SES[\,[\![P]\!]\,]\langle r\rangle \\
[\![P > (?x)Q]\!] & \stackrel{\text{def}}{=} & \underline{x > Q}[\,[\![P]\!]\,]\langle\lceil fn(Q) \setminus \{x\}\rceil\rangle \\
[\![P \mid Q]\!] & \stackrel{\text{def}}{=} & [\![P]\!] \mid [\![Q]\!] \\
[\![(\nu w)P]\!] & \stackrel{\text{def}}{=} & (\nu w)[\![P]\!]
\end{array}$$

It is worth stressing that if one is interested in analyzing a finite set of CaSPiS processes through the encoding to the algebra of designs and the transformation of the corresponding graphs, then the resulting algebra will have a finite number of edge and design labels, determined by the set of sub-processes of those of interest. Instead, to be able to accommodate the encoding of all possible CaSPiS processes, denumerable sets of labels are needed.

Notably, structural congruence amounts to design equivalence, i.e. equivalent processes are mapped into isomorphic graphs.

Proposition 1 *For any $Q, R \in \mathscr{P}$ we have $P \equiv_{\mathrm{C}} Q$ iff $[\![P]\!] \equiv_{\mathrm{D}} [\![Q]\!]$.*

4.1 Transformation rules for CaSPiS reduction semantics

Given the encoding of CaSPiS processes as terms of the algebra of designs, and any suitable model of the algebra in terms of a class of graphs (like those presented in Sections 2.1 and 2.2), it is natural to try to lift the reduction semantics of CaSPiS, through these encodings, to a corresponding notion of transformation over the resulting graphs. Ideally, we would like to translate the reduction rules of Definition 8 to ordinary graph transformation rules, in order to exploit the rich theory of graph transformation and the corresponding analysis and verification tools, also accounting for concurrency aspects.

However, this is not possible in a direct way. In fact, the reduction rules of CaSPiS include suitable contexts in the left- and right-hand sides, which can be instantiated in arbitrary ways to match a subterm of the process to be reduced. In other words, each reduction rule can be considered as a rule schema, summarizing the common shape of infinitely many similar rules, obtained by consistently replacing the contexts with suitable terms. Quite obviously, if we are interested in reducing a single process (or a finite set of processes), we need to consider only a finite set of instances of the rules.

In Figures 6, 7 and 8 we depicted the graph transformation rule schemata corresponding to the reduction rules (*ServiceSync*), (*SessionSync*) and (*PipelineSync*), using the side-view of designs discussed in Section 2.2. Comparing them with the corresponding reduction rules in

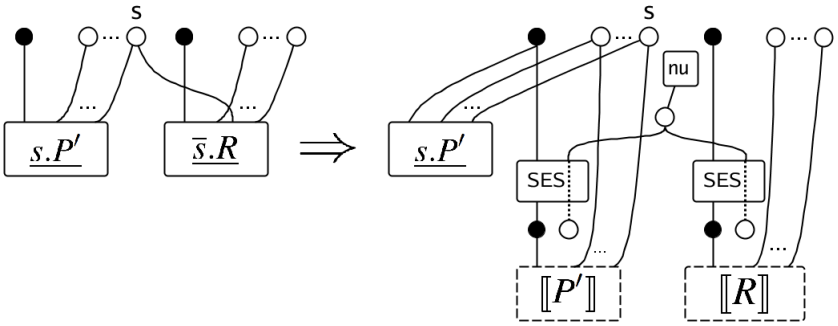

Figure 6: Rule (*ServiceSync*)

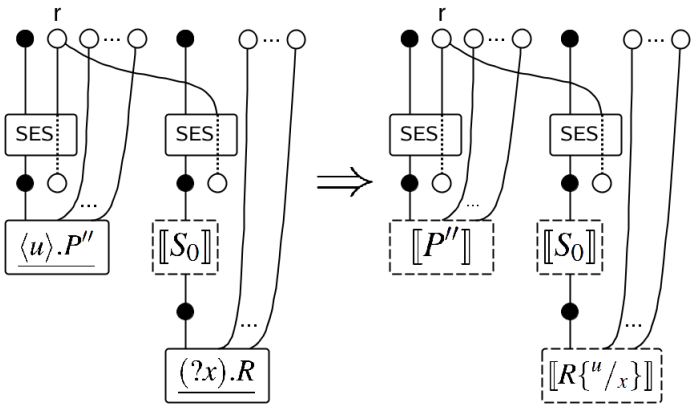

Figure 7: rule (*SessionSync*)

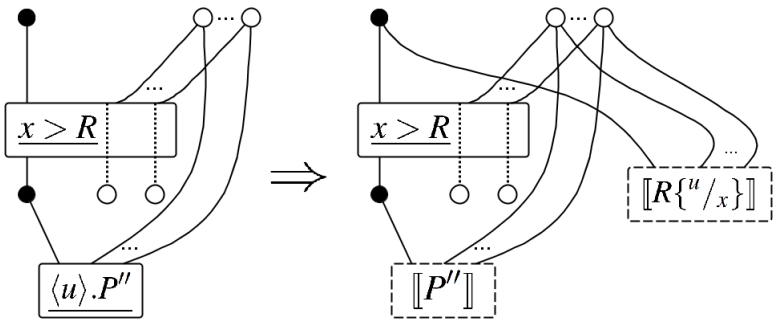

Figure 8: Rule (*PipelineSync*)

Fig. 5, we can note that: 1) we can now omit to specify the static context C under which the interacting redexes are found, because the left-hand side of a graph transformation rule can always be applied in larger graphs; 2) for the same reason, we can safely omit idle items that run in parallel, like process P' from rules (*SessionSync*) and (*PipelineSync*) in Fig. 5; 3) but we must still account for the presence of any admissible static session-immune context S_0 in rules (*SessionSync*), (*SessionSyncRet*) and (*PipelineSyncRet*), because it constrains the applicability of the rule (in general $[\![S_0]\!]$ can be a chain of pipeline-labeled boxes of arbitrary length, possibly 0). Even if we did not work out the corresponding definitions, we identified two graph transformation frameworks which can provide the means to turn each such rule schema into a collection of graph rules, whose overall effect would be the expected one when applied to a graph representing a CaSPiS process.

Synchronized Hyperedge Replacement. In the SHR approach [FHL+06], the parallel application of a set of rules to a graph is controlled by a synchronization mechanism which requires a consistency check among the redex boundaries of the involved rules. This mechanism can be used to build (standard) rules with unbound left-hand sides, starting from a finite set of rules. Therefore a CaSPiS rule schema could be implemented by a set of SHR rules, which should be able to induce the set of all its instantiations.

Graph Transactions. The notion of graph transaction proposed in [BCD+08] is based on a notion of "unstable" graph items. A transaction is a minimal derivation starting and ending in graphs not containing unstable items, up to shift equivalence, and the operational semantics of a transactional graph transformation system includes only derivations that are made of transactions. Therefore a CaSPiS rule schema could be translated into a collection of rules which simulate the navigation of the process in order to identify an occurrence of the left hand side. This can be done by generating unstable items in the graph: their presence conceptually inhibits the application of other rules in parallel. When the left pattern is recognized and the effect of the rule is applied, such unstable elements are deleted, resulting in the commitment of the transaction.

The study of these possible translations of CaSPiS rule schemata is an interesting topic for future research.

5 Conclusions

In this paper we have shown the main issues regarding the graphical encoding of a sophisticated process calculus with inherently hierarchical features. The encoding of processes can be written quite smoothly by exploiting a recently proposed algebra of graphs with nesting (see Definition 9), and it can be shown to preserve and respect the structural congruence of processes. On the other hand, the encoding of reduction rules as ordinary graph transformation rules requires some ingenuity, because the redexes can require the traversal/inspection of an unbound number of nesting levels due to the presence of static session-immune contexts in the rules of Fig. 5.

The main methodological innovation of the paper, with respect to other proposals of encoding process algebras into graph transformation systems, resides in the identification of an intermediate

algebra of designs, which bridges the gap between the syntax of the process calculus and the set theoretical definition of the graphs. A direct translation of CaSPiS processes to, for example, gs-graphs, would be possible but more cumbersome. Furthermore, a sound and complete interpretation of the algebra into a class of graphs can be reused for different process calculi. For example, besides the top- and side-view graphs discussed in the paper, another natural graph model for the algebra are Milner's bigraphs [Mil06], which are naturally endowed with a notion of embedding and of linking.

The ultimate motivation in equipping CaSPiS with a graph transformation operational semantics is to exploit the rich theory of graph transformation and corresponding tools for the analysis and verification of relevant properties of CaSPiS processes. The intermediate design algebra provides one additional framework for such analysis, which could be performed by exploiting tools directly based on the algebra, which are currently under development (see http://www.albertolluch.com/adr2graphs/).

Acknowledgements: We want to thank Fabio Gadducci, Alberto Lluch Lafuente, Daniele Terreni and Liang Zhao for many interesting discussions and exchanges of ideas regarding the graphical encoding of CaSPiS.

Bibliography

[AG98] A. Asperti, S. Guerrini. *The Optimal Implementation of Functional Programming Languages*. Cambridge University Press, 1998.

[BBDL08] M. Boreale, R. Bruni, R. De Nicola, M. Loreti. Sessions and Pipelines for Structured Service Programming. In Barthe and de Boer (eds.), *FMOODS 2008*. LNCS 5051, pp. 19–38. Springer, 2008.

[BCD+08] P. Baldan, A. Corradini, F. Dotti, L. Foss, F. Gadducci, L. Ribeiro. Towards a Notion of Transaction in Graph Rewriting. In Bruni and Varró (eds.), *International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2006)*. ENTCS 211. Elsevier, 2008.

[BCG+] R. Bruni, A. Corradini, F. Gadducci, A. Lluch Lafuente, U. Montanari. On GS-Monoidal Theories for Graphs with Nesting. Submitted for publication.

[BEG+87] H. Barendregt, M. van Eekelen, J. Glauert, J. Kennaway, M. Plasmeijer, M. Sleep. Term graph reduction. In *PARLE'87*. LNCS 259, pp. 141–158. Springer, 1987.

[BGL10a] R. Bruni, F. Gadducci, A. Lluch Lafuente. An Algebra of Hierarchical Graphs. In *Proceedings TGC 2010*. LNCS. Springer, 2010. To appear.

[BGL10b] R. Bruni, F. Gadducci, A. Lluch Lafuente. An Algebra of Hierarchical Graphs and its Application to Structural Encoding. *Scientific Annals of Computer Science*, 2010. To appear.

[BGL10c] R. Bruni, F. Gadducci, A. Lluch Lafuente. A Graph Syntax for Processes and Services. In Jianwen and Laneve (eds.), *Proceedings WS-FM 2009*. LNCS. Springer, 2010. To appear.

[BL05] R. Bruni, I. Lanese. On Graph(ic) Encodings. In Koenig et al. (eds.), *Proceedings of Dagstuhl Seminar n. 04241, Graph Transformations and Process Algebras for Modeling Distributed and Mobile Systems*. Pp. 23–29. 2005.

[BLMT08] R. Bruni, A. Lluch Lafuente, U. Montanari, E. Tuosto. Style Based Architectural Reconfigurations. *Bulletin of the European Association for Theoretical Computer Science (EATCS)* 94:161–180, February 2008.

[BMM06] R. Bruni, H. Melgratti, U. Montanari. Event Structure Semantics for Nominal Calculi. In Baier and Hermanns (eds.), *CONCUR 2006*. LNCS 4137, pp. 295–309. Springer, 2006.

[CG99] A. Corradini, F. Gadducci. An Algebraic Presentation of Term Graphs, via GS-Monoidal Categories. *Applied Categorical Structures* 7:299–331, 1999.

[CMR94] A. Corradini, U. Montanari, F. Rossi. An Abstract Machine for Concurrent Modular Systems: CHARM. *Theoretical Computer Science* 122(1-2):165–200, 1994.

[DHP02] F. Drewes, B. Hoffmann, D. Plump. Hierarchical Graph Transformation. *Journal on Computer and System Sciences* 64(2):249–283, 2002.

[FHL+06] G. L. Ferrari, D. Hirsch, I. Lanese, U. Montanari, E. Tuosto. Synchronised Hyperedge Replacement as a Model for Service Oriented Computing. In Boer et al. (eds.), *FMCO 2005*. LNCS 4111, pp. 22–43. Springer, 2006.

[FM00] G. L. Ferrari, U. Montanari. Tile Formats for Located and Mobile Systems. *Information and Computation* 156(1-2):173–235, 2000.

[Gad03] F. Gadducci. Term Graph Rewriting for the pi-Calculus. In Ohori (ed.), *APLAS 2003*. LNCS 2895, pp. 37–54. Springer, 2003.

[KCM06] D. Kitchin, W. R. Cook, J. Misra. A Language for Task Orchestration and Its Semantic Properties. In Baier and Hermanns (eds.), *CONCUR 2006*. LNCS 4137, pp. 477–491. Springer, 2006.

[Mil06] R. Milner. Pure bigraphs: Structure and dynamics. *Information and Computation* 204(1):60–122, 2006.

[MP95] U. Montanari, M. Pistore. Concurrent semantics for the pi-calculus. *Electr. Notes Theor. Comput. Sci.* 1, 1995.

[Roz97] G. Rozenberg (ed.). *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*. World Scientific, 1997.

[Ter08] D. Terreni. Computational models based on hierarchical graphs: bigraphs and cogs-graphs. Master thesis, Dipartimento di Informatica, Università di Pisa. 2008.