A Graph Syntax for Processes and Services*

Roberto Bruni, Fabio Gadducci, and Alberto Lluch Lafuente

Department of Computer Science, University of Pisa
bruni,gadducci,lafuente@di.unipi.it

Abstract. We propose a class of hierarchical graphs equipped with a simple algebraic syntax as a convenient way to describe configurations in languages equipped with inherently hierarchical features such as sessions, fault-handling scopes, transactions or locations. The graph syntax can be seen as an intermediate representation language, that facilitates the encoding of algebraic specifications and, in particular, of process calculi, since it provides primitives for nesting, name restriction and parallel composition. The syntax is based on an algebraic presentation of graphs that faithfully characterises families of hierarchical graphs, meaning that each term of the language identifies uniquely an equivalence class of graphs (modulo graph isomorphism). Proving soundness and correctness of an encoding (i.e. proving that structurally congruent processes are mapped to isomorphic graphs) is then facilitated and can be done by structural induction. Summing up, the graph syntax facilitates the definition of faithful encodings, yet allowing a precise visual representation. We instantiate our proposal by offering the graphical encoding of a workflow language and of a service-oriented calculus.

1 Introduction

As exemplified by a large literature, graphs offer a convenient ground for the specification and analysis of modern software systems with features such as distribution, concurrency and mobility. Among the graph-based formalisms used for such purposes, we recall those based on traditional Graph Transformation [1], Bigraphical Reactive Systems [2] and Synchronized Hyperedge Replacement [3]. Using any of such approaches to build a graphical representation of an existing language involves two major challenges: encoding states and encoding the operational semantics. A correct state encoding should map structurally equivalent states into equivalent (typically isomorphic) graphs. In addition, the state encoding should also facilitate the encoding of the operational semantics, which typically means mimicking term rewrites via suitable graph rewrites.

The use of isomorphism as graph equivalence has several advantages. Among others, it offers an intuitive normal form representation for systems, it allows to reuse results or techniques from graph theory for solving specific problems (system equivalence via graph isomorphism), and it enables traditional graph transformations, which have (sub)graph isomorphism at the base of the matching

* Research supported by the EU FET integrated project Sensoria, IST-2005-016004.

mechanism used for the application of rules. Of course, capturing equivalence of configurations via graph isomorphism is not always possible or convenient. In such cases additional axioms must be added to isomorphism. The reuse of standard graph transformation techniques in such situations is not new. For instance one typical solution is to consider (equivalence classes of) normal forms for graphs and a set of confluent and terminating rewrite rules that reduce graphs into the normal form of their equivalence class.

When the language is a process calculus, the encoding of a state is facilitated by its algebraic structure (since processes are terms) and it is typically defined inductively on such structure. However, the syntax of graph formalisms is often not provided with suitable features for names, name restrictions or hierarchical aspects. Typical solutions consist in developing ad-hoc algebraic syntaxes that are often significantly different from the syntax of process calculi. They require advanced skills and are based on sophisticated techniques involving set-theoretic definition of graphs with interfaces (e.g. [1, 4]), enriched type systems (e.g. [5, 6]) or representing hierarchies as trees (e.g. [2, 4]) and may result in layered representations and complex correctness proofs.

Our goal is to develop a technique for simplifying the definition of state encodings into graphical structures and the proof of their correctness and such that the associated graph rewriting rules are automatically determined from the state encoding and the original operational semantics.

In a companion paper [7] we present some results that help in overcoming some of the drawbacks mentioned above. The proposal described there fills the gap between the different levels of abstraction at which process calculi and graphical structures reside by introducing a specification formalism made of an algebra of hierarchical graphs, equipped with a sound and complete set of axioms equating two terms whenever they represent essentially the same hierarchical graph. The graph algebra is equipped with primitives and mechanisms for dealing with names, name restriction, parallel composition and, most importantly, nesting in the same way as they are used in process calculi. In particular, the nesting mechanism allows for easily defining graphical presentations for process calculi with inherently hierarchical aspects such as sessions, transactions or locations: features of fundamental relevance, e.g. in the area of service-oriented computing. Besides facilitating the visual specification of processes, the graph algebra simplifies the proofs of correctness: the algebraic structure of both states and graphs enables proofs by structural induction.

In this paper we validate our ideas by using our graph algebra (presented in Section 2) to encode the configurations of (process) calculi with service-inherent features that have a certain *hierarchical nature* such as sessions, transactions or locations. In particular, we provide novel, correct graphical encodings for two languages. The first one (Section 3) is a simple workflow language, vaguely reminiscent of BPEL: it is used for showing the basic features of the graph syntax and getting the reader acquainted with the approach. The second example (Section 4) regards a sophisticated calculus for the description of service-oriented applications, namely, CaSPiS [8], whose features pose further challenges to visu-

alisation, due to the interplay of name handling, nested sessions and a pipeline operator. We remark that the technique we propose can be transferred to other calculi as well, as witnessed by other available encodings mentioned in Section 5.

2 An algebra of hierarchical graphs

We offer an overview of our algebra of typed, hierarchical (hyper)graphs[1] that we call *designs*, referring to [7] for a detailed presentation.

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

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

where l *and* L *are respectively drawn from vocabularies* $\mathcal{T}$ *and* $\mathcal{NT}$ *of edge and design labels,* $\mathcal{N}$ *is the set of nodes,* $x \in \mathcal{N}$ *and* $\overline{x} \in \mathcal{N}^*$*.*

The algebraic reading is as usual, where each syntactical category and vocabulary is considered as a sort and productions are read as functions. This allows us, for instance, to consider open terms (i.e. terms with typed variables), useful for defining encodings by means of derived operators.

Terms generated by $\mathbb{G}$ are (possibly hierarchical) graphs: $\mathbf{0}$ represents the empty graph, x is a discrete graph containing node x only, $l(\overline{x})$ is a flat graph formed by an l-labelled edge attached to nodes $\overline{x}$ (the i-th tentacle to the i-th node in $\overline{x}$), $\mathbb{G} \mid \mathbb{H}$ is the graph resulting from the parallel composition of graphs $\mathbb{G}$ and $\mathbb{H}$ (their disjoint union up to common nodes), $(\nu x)\mathbb{G}$ is graph $\mathbb{G}$ after restricting node x, and $\mathbb{D}\langle\overline{x}\rangle$ is a graph formed by attaching design $\mathbb{D}$ to nodes $\overline{x}$ (the i-th node in the interface of $\mathbb{D}$ to the i-th node in $\overline{x}$). A term $L_{(\overline{x})}[\mathbb{G}]$ represents a design of type L, with body graph $\mathbb{G}$ and exposing nodes $\overline{x}$ in its interface. We use the notation $L_{\langle\overline{y}\rangle}[\mathbb{G}\{^{\overline{y}}/_{\overline{x}}\}]$ as a shorthand for $L_{(\overline{x})}[\mathbb{G}]\langle\overline{y}\rangle$, where $\{^{\overline{y}}/_{\overline{x}}\}$ denotes name substitution.

Example 1. Let $a \in \mathcal{T}$, $A \in \mathcal{NT}$, $u, v, w, x, y \in \mathcal{N}$. We depict in Fig. 1 some terms of our algebra: u (top-left), $a(u, v)$ (top-second), $a(u, w) \mid a(w, v)$ (top-third), $(\nu w)(a(u, w) \mid a(w, v))$ (top-right), and $A_{(u,v)}[(\nu w)(a(u, w) \mid a(w, v))]\langle x, y\rangle$ (bottom-left), also abbreviated as $A_{\langle x,y\rangle}[(\nu w)(a(x, w) \mid a(w, y))]$ Nodes are represented by circles, edges by boxes, and designs by dotted boxes. The first tentacle of an edge is represented by a plain arrow with no head, while the second one is denoted by a normal arrow. The rest are restricted or interface nodes.

Note that this representation is informal and aims at offering an intuitive visualisation. In [7] we offer the formal representation of the same terms. Figure 1 also includes term $A_{\langle x,y\rangle}[a(x, w) \mid a(w, y)] \mid A_{\langle y,x\rangle}[a(y, w) \mid a(w, x)]$ (bottom-right), where two designs are composed in circle by attaching them symmetrically to x and y, and where they share (as a common name) node w.

[1] Hypergraphs extend graphs with (hyper)edges of arbitrary cardinality (not just 2).

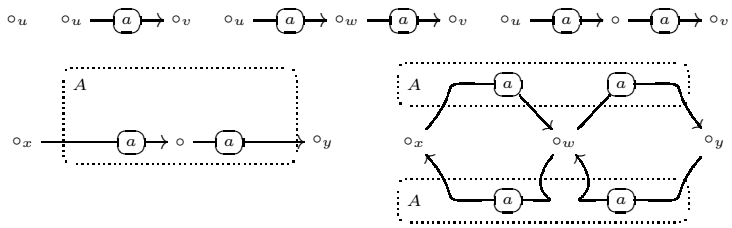

Fig. 1. Some terms of the graph algebra

Sort $\mathbb{D}$ is partitioned over the set $\mathcal{NT} = \{L_1, \ldots, L_n\}$, i.e., we consider sorts $L_1, \ldots, L_n$ and a membership predicate $\mathbb{D} : L$ that holds whenever $\mathbb{D} = L_{(\overline{x})}[\mathbb{G}]$ for some $\overline{x}$ and $\mathbb{G}$. Thus, design labels play the role of design (sub-)types. Likewise, we consider the set of nodes $\mathcal{V}$ to be partitioned over different sorts.

Interface and restricted nodes lead us to the usual concept of *free* nodes.

Definition 2 (free nodes). *The* free *nodes of a design or graph are denoted by the function fn, defined as follows*

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

where $|\overline{x}|$ denotes the set of elements of a vector $\overline{x}$.

Each label of $\mathcal{T}$ and $\mathcal{NT}$ has a fixed arity and for each rank a fixed node type. Intuitively, the typed arity of a label denotes the ordered and typed tentacles of edges with that label. We say that a design (or a graph) is well-typed if for each occurrence of a typed operator $L_{(\overline{x})}[\mathbb{G}]$ we have the (vectors of) types of $\overline{x}$ and L coincide, and similarly for typed operators $\mathbb{D}\langle\overline{x}\rangle$ and $l(\overline{x})$. In the following we assume all substitutions respect typing (and thus preserve well-formedness).

Definition 3 (well-formedness). *A design or graph is* well-formed *if (1) it is well-typed; (2) for each occurrence of design $L_{(\overline{x})}[\mathbb{G}]$ we have $|\overline{x}| \subseteq fn(\mathbb{G})$; and (3) for each occurrence of graph $L_{(\overline{x})}[\mathbb{G}]\langle\overline{y}\rangle$ the mapping ${}^{\overline{y}}/_{\overline{x}}$ is a bijection.*

Intuitively, the restriction on the mapping ${}^{\overline{y}}/_{\overline{x}}$ forbids two distinct nodes at the higher level to be mapped to the same node in $\mathbb{G}$. This is needed for avoiding implicit name fusions (equivalently, node coalescing) as the result of applying a *flattening* axiom, as shown below.[2] From now on we restrict our attention to well-formed designs: all axioms will preserve well-formedness and all derived operators used for the encodings will be well-formed.

In order to have a notion of syntactically equivalent designs (i.e., to consider designs up to isomorphism) the algebra includes the structural graph axioms of [9] such as AC1 for $\mid$ (with identity $\mathbf{0}$) and name extrusion (respectively,

[2] The restriction can be dropped, but this requires an algebra with explicit fusion operators (of no use in this paper).

axioms DA1–DA3 and DA4–DA6). In addition, it includes axioms to α-rename bound nodes (DA7–DA8), node extrusion for designs (DA9), and an axiom for avoiding the addition of a node to a graph where that same node is already free (DA10).

Definition 4 (design axioms). *The structural congruence* $\equiv_d$ *over well-formed designs and graphs is the least congruence satisfying*

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

where in the axiom (DA7) *the involved substitution is required to be abijection, in order to avoid node coalescing.*

We call a graph *flat* whenever there is no design in its body. Flattening a design is done by a kind of hyper-edge replacement [10] in the form of axioms that are sometimes useful to be included in the structural congruence.

Definition 5 (flattening axiom). *A flattening axiom* $\mathsf{flat_L}$ *for some design label* L *is of the form* $L_{\langle\overline{y}\rangle}[\mathbb{G}] \equiv \mathbb{G}$ *(i.e.* $L_{(\overline{x})}[\mathbb{G}]\langle\overline{y}\rangle \equiv \mathbb{G}\{^{\overline{y}}/_{\overline{x}}\}$*).*

Example 2. Suppose that we want to characterise the set of a-labelled, acyclic, and connected sequences. We can define an algebra with an element α in the sequence, and a binary sequential composition $_;_$. Both are derived operators defined by $\alpha \stackrel{\text{def}}{=} A_{(u,v)}[a(u,v)]$ and $X;Y \stackrel{\text{def}}{=} A_{(u,v)}[(\nu w)(X\langle u,w\rangle \mid Y\langle w,v\rangle)]$, where X and Y have type A. Clearly, the algebra as such constructs hierarchical sequences, where e.g. $(\alpha;(\alpha;\alpha))\langle x,y\rangle$ and $((\alpha;\alpha);\alpha)\langle x,y\rangle$ are not equivalent graphs due to different nestings. Introducing $\mathsf{flat_A}$ in the algebra, instead, we have that the two former terms are identified, and intuitively correspond to the normal form $(\nu w_1,w_2)(a(x,w_1) \mid a(w_1,w_2) \mid a(w_2,y))$.

This example illustrates the two roles of the nesting operator: as a means to enclose a graph and as sort of typed interface to enable disciplined graph compositions. The presence of flattening axioms makes the first role implicit.

3 Graphical interpretation of workflows

This section presents the use of our graph-based language for algebraic workflow specifications. We consider a minimal language that includes, nonetheless, typical workflow ingredients and offers an attractive presentation of our technique.

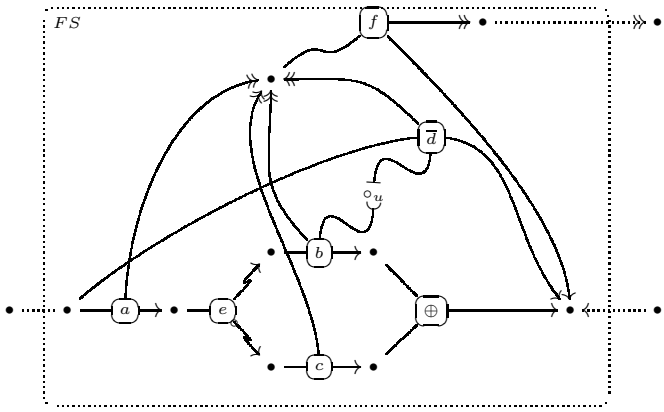

Fig. 2. A simple workflow

A simple language for workflows. Our simple workflow language considers workflows of activities that can be composed in sequence, parallel or by branching. The control flow is restricted to one entry point and two exit points: one for the successful completion and one for error raising. Error-handling activities and error scopes are also considered. In addition, we consider synchronisation links as present in some workflow languages like BPEL.

Figure 2 depicts a very simple example including all the ingredients: we see a main composed flow with an event handler f. The main flow starts with activity a, it performs activity d in parallel with a choice between activities b and c (depending on e). A dependency is imposed between b and d.

More precisely, the syntax of the workflow language is formally defined below.

Definition 6 (workflow). *Let $\mathcal{A}$ be a set of* activity names*, $\mathcal{E}$ be a set of* expressions*, and $\mathcal{U}$ be a set of* synchronisation points*. A* workflow *F is a term generated by the following syntax (where $a \in \mathcal{A}$, $e \in \mathcal{E}$ and $u \in \mathcal{U}$):*

$$\begin{array}{lcl} F & ::= & A \mid F;F \mid \mathsf{if}\ e\ \mathsf{then}\ F\ \mathsf{else}\ F \mid F|F \mid \mathsf{try}\ F\ \mathsf{catch}\ F \\ A & ::= & a \mid a(u) \mid a[u] \end{array}$$

More precisely, a is an asynchronous activity, $a(u)$ is an activity of type a with source link u, $a[u]$ is an activity of type a with target link u, $G;H$ is the sequential composition of structured flows G and H, $\mathsf{if}\ e\ \mathsf{then}\ G\ \mathsf{else}\ H$ introduces a binary branch, i.e. a choice between flows G and H depending on the evaluation of e, $G \mid H$ is the parallel composition of structured flows G and H, and $\mathsf{try}\ G\ \mathsf{catch}\ H$ inserts a new error scope for flow G with fault handler H. As an example, the workflow of Fig. 2 corresponds to the following workflow term $\mathsf{try}\ d[u] \mid a; (\mathsf{if}\ e\ \mathsf{then}\ b(u)\ \mathsf{else}\ c)\ \mathsf{catch}\ f$.

We consider a structural congruence that basically models the fact that sequential composition is associative and parallel composition is associative and commutative. This is formally defined as follows.

Definition 7 (structural congruence). *The* structural congruence *for workflows is the relation* $\equiv_w \subseteq \mathcal{W} \times \mathcal{W}$*, closed under workflow construction and inductively generated by the following set of axioms*

$$G;(H;I) \equiv (G;H);I \quad (w\mathsf{A1}) \quad G \mid (H \mid I) \equiv (G \mid H) \mid I \quad (w\mathsf{A2}) \quad G \mid H \equiv H \mid G \quad (w\mathsf{A3})$$

Workflow encoding. The encoding of our workflow language is depicted in Fig. 3. We explain our graphical notation of design operations in detail here. An edge is represented by a rounded box with its label inside. We see that the types of nodes we use are $\bullet$ and $\circ$ which respectively represent control flow and synchronisation links. We use an encircled circle $\circledcirc$ to denote an argument of type $\circ$ in the encoding of activities. In case of encodings with more than one argument for a type we denote the argument order explicitly by subscripting (e.g. the two arguments of type F in the encoding of failure handling, sequence, branching and parallel composition). Terminal edge types include $a \in \mathcal{A}$, $e \in \mathcal{E}$ and $\oplus$ which are respectively used to represent activities, expressions and to denote branch closing (xor join). To improve visualization impact, we use different kinds of arrows to denote tentacles. A plain tentacle represents an entry point, while a simple arrow indicates an ordinary exit point. A double arrow indicates the fault exit point. In a conditional choice, the *then* branch is denoted with an ordinary arrow, while the *else* branch is denoted with an arrow with a small circle on its tail. A bar-ended tentacle denotes the synchronisation link of an activity. Link sources and targets are respectively represented by concave and bar-ended tentacles. Finally, we consider the non-terminal type F to stand for workflows. Dotted arrows denote node exposure and an enclosing dotted box represents a design with its type on the upper-left corner. All nodes are bound except for the argument names (as in the case of synchronisation points in the encoding of activities). Most of these ingredients can be found in Fig. 3.

The formal definition by means of our graph algebra is as simple as follows.

Definition 8 (workflow interpretation). *The interpretation of the operators of the workflow language over the design algebra is given by:*

$$\begin{aligned}
a &\stackrel{\mathbf{def}}{=} F_{(in,out,fail)}[a(in,out,fail)] \\
a(u) &\stackrel{\mathbf{def}}{=} F_{(in,out,fail)}[a(in,out,fail,u)] \\
a[u] &\stackrel{\mathbf{def}}{=} F_{(in,out,fail)}[\overline{a}(in,out,fail,u)] \\
G;H &\stackrel{\mathbf{def}}{=} F_{(in,out,fail)}[(\nu\, mid)(G\langle in,mid,fail\rangle \mid H\langle mid,out,fail\rangle)] \\
\mathsf{if}\ e\ \mathsf{then}\ G\ \mathsf{else}\ H &\stackrel{\mathbf{def}}{=} (\nu\, th1,th2,el1,el2)F_{(in,out,fail)}[e(in,th1,el1) \\
&\qquad \mid G\langle th1,th2,fail\rangle \mid H\langle el1,el2,fail\rangle \mid \oplus(th2,el2,out)] \\
G \mid H &\stackrel{\mathbf{def}}{=} F_{(in,out,fail)}[G\langle in,out,fail\rangle \mid H\langle in,out,fail\rangle] \\
\mathsf{try}\ G\ \mathsf{catch}\ H &\stackrel{\mathbf{def}}{=} F_{(in,out,fail)}[(\nu mid)FS_{\langle in,out,fail\rangle}[G\langle in,out,mid\rangle \\
&\qquad \mid H\langle mid,out,fail\rangle]]
\end{aligned}$$

together with axiom $\mathsf{flat}_{\mathsf{F}}$*.*

It is easy to prove that structural congruence amounts to design equivalence, i.e., equivalent workflows amount to equivalent graphs. Note that thanks to

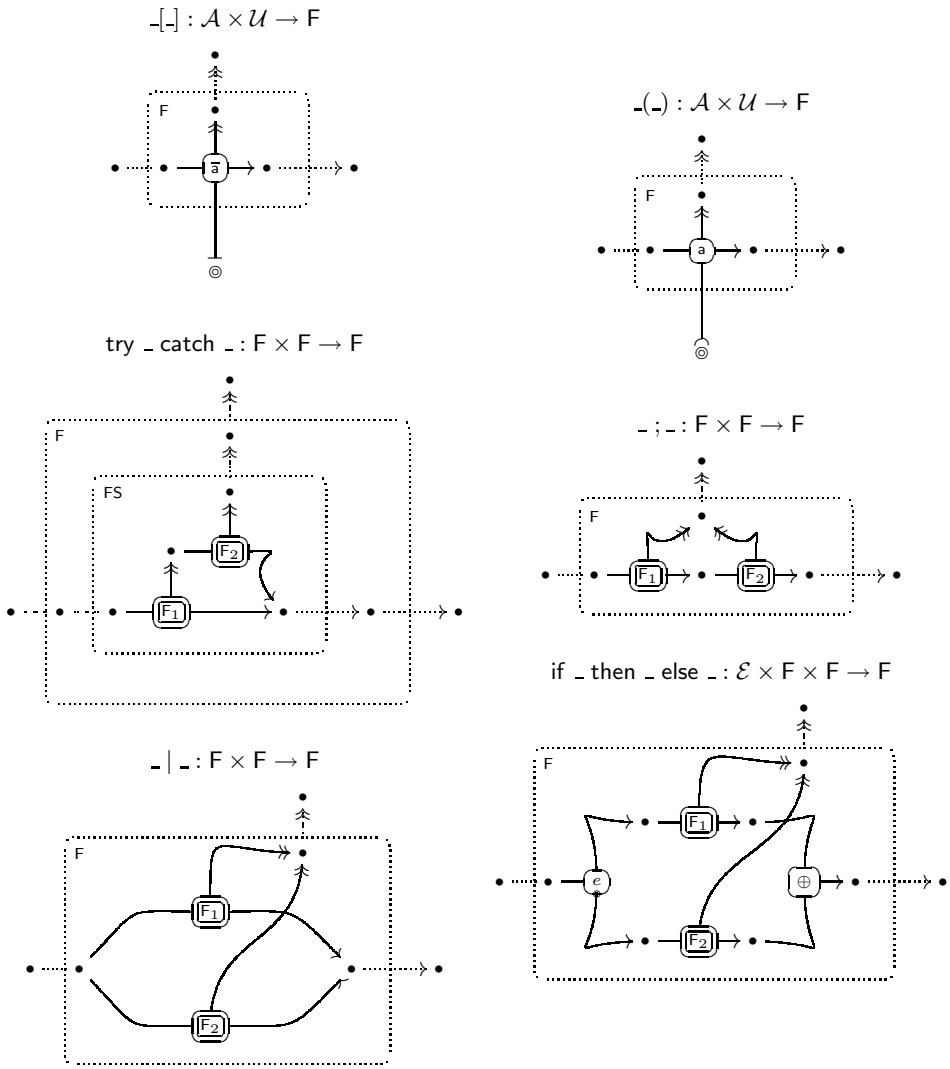

Fig. 3. Graphical encoding of a simple workflow language

axiom DA9 node restriction can be equivalently placed in the innermost design (see $G; H$), at the topmost level (see if e then G else H) or at any intermediate level of nesting (see try G catch H).

Proposition 1. *For any two workflows G and H we have $G \equiv_w H$ iff $G \equiv_d H$.*

4 Graphical interpretation of CaSPiS

This section presents the graphical representation of CaSPiS by defining each CaSPiS syntactic constructor as a derived operator of our graph algebra. We offer a minimal presentation of CaSPiS and refer to [8] for a more detailed description.

Definition 9 (CaSPiS syntax). *Let $\mathcal{R}$ be a set of session names, $\mathcal{S}$ a set of service names and $\mathcal{V}$ a set of value names. A CaSPiS process P is a term generated by the syntax*

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

where $s \in \mathcal{S}$, $r \in \mathcal{R}$, $u \in \mathcal{V}$, $w \in \mathcal{V} \cup \mathcal{R}$ and x is a value variable.

Service definitions and invocations are written like input and output prefixes in CCS. Thus $s.P$ defines a service s that can be invoked by $\overline{s}.Q$.

Synchronisation of $s.P$ and $\overline{s}.Q$ leads to the creation of a new session, identified by a fresh name r that can be viewed as a private, synchronous channel binding caller and callee. Since client and service may be far apart, a session naturally comes with two sides, written $r \triangleright P$, with r bound somewhere above them by (νr). Rules governing creation and scoping of sessions are based on those of the restriction operator in the π-calculus. Note that nested invocations to services will yield separate sessions and thus hierarchies of nested sessions.

When two partner sides $r \triangleright P$ and $r \triangleright Q$ are deployed, intra-session communication is done via input and output actions $\langle u \rangle$ and $(?x)$: values produced by P can be consumed by Q, and vice-versa. This permits interaction patterns more complex than the typical *one-way* and *request-response* of web services.

Values can be returned outside a session to the enclosing environment using the return operator $\langle \cdot \rangle^{\uparrow}$. Return values can be consumed by other sessions, or used to invoke other services, to start new activities. This is achieved using the pipeline operator $P > Q$. Here, a new instance of process Q is activated each time P emits a value that Q can consume. Notably, the new instance of Q will run within the same session as $P > Q$, not in a fresh one.

CaSPiS processes can be considered up to the structural congruence $\equiv_c$.

Definition 10 (CaSPiS congruence). *The structural congruence $\equiv_c$ is the least congruence induced by the following laws*

$$\begin{array}{rcll} P \mid (Q \mid R) \equiv (P \mid Q) \mid R) & & \text{(CA1)} & \\ P \mid Q \equiv Q \mid P & & \text{(CA2)} & \\ P \mid \mathbf{0} \equiv P & & \text{(CA3)} & \\ (\nu n)(\nu m)P \equiv (\nu m)(\nu n)P & & \text{(CA4)} & \\ (\nu n)\mathbf{0} \equiv \mathbf{0} & & \text{(CA5)} & \\ P \mid (\nu n)Q \equiv (\nu n)(P \mid Q) & \text{if } n \notin fn(P) & \text{(CA6)} & \\ ((\nu n)Q) > P \equiv (\nu n)(Q > P) & \text{if } n \notin fn(P) & \text{(CA7)} & \\ A.(\nu n)P \equiv (\nu n)A.P & \text{if } n \notin A & \text{(CA8)} & \\ r \triangleright (\nu n)P \equiv (\nu n)r \triangleright P & \text{if } n \neq r & \text{(CA9)} & \\ (\nu n)P \equiv (\nu m)P\{^m/_n\} & \text{if } m \notin fn(P) & \text{(CA10)} & \\ (?x).P \equiv (?y).P\{^y/_x\} & \text{if } y \notin fn(P) & \text{(CA11)} & \end{array}$$

CaSPiS encoding. We first define the alphabets of edge labels and nodes. The set $\mathcal{NT}$ of design labels is composed by P, S, D, I, F and T which respectively stand for Parallel processes, Sessions, service Definitions, service Invocations and pipes (From and To). Sort T is further partitioned over $2^{\mathcal{V}\cup\mathcal{R}}$ (denoting each subsort as T^N) to deal with a common problem when encoding replicated processes (the target process of a pipe is implicitly replicated for each value generated by the source). The set $\mathcal{T}$ of edge labels contains def (service definition), inv (service invocation), in (input), out (output) and ret (return). The node sorts considered are $\circ$ (channels), $\bullet$ (control points), $*$ (service names, i.e. $\mathcal{S}$) and $\square$ (values, i.e. $\mathcal{V}$). We assume that for each session name r there is a channel node.

The graphical representation of each design and edge label and their respective types can be found in Fig. 4. For instance, designs of type P are all of the form $P_{(p,t,o,i)}[G]$ where p is the control point representing the process start of execution, t is the returning channel, i is the input channel and o is the output channel. Designs of type D and I only expose the starting point of execution.

Definition 11 (CaSPiS interpretation). *The interpretation of CaSPiS constructors as derived operators of the design algebra is given by*

$$
\begin{aligned}
s.Q &\stackrel{\text{def}}{=} P_{(p,t,o,i)}[\, t|o|i|\, D_{\langle p\rangle}[\,(\nu q,t',o',i')(\mathsf{def}(p,s,q)|Q\langle q,t',o',i'\rangle)\,]\,] \\
\overline{s}.Q &\stackrel{\text{def}}{=} P_{(p,t,o,i)}[\, t|o|i|\, I_{\langle p\rangle}[\,(\nu q,t',o',i')(\mathsf{inv}(p,s,q)|Q\langle q,t',o',i'\rangle)\,]\,] \\
r \triangleright Q &\stackrel{\text{def}}{=} P_{(p,t,o,i)}[\, t|i|\, S_{\langle p,o\rangle}[\, Q\langle p,o,r,r\rangle\,]\,] \\
Q > R &\stackrel{\text{def}}{=} P_{(p,t,o,i)}[\,(\nu q,m)(\, F_{\langle p,t,m,i\rangle}[\, Q\langle p,t,m,i\rangle\,] \\
&\qquad\qquad |\, T^{fn(R)}_{\langle m\rangle}[\,(\nu q,t',o')R\langle q,t',o',m\rangle\,]\,)\,] \\
Q|R &\stackrel{\text{def}}{=} P_{(p,t,o,i)}[\, Q\langle p,t,o,i\rangle|R\langle p,t,o,i\rangle\,] \\
(\nu w)Q &\stackrel{\text{def}}{=} P_{(p,t,o,i)}[(\nu w)Q\langle p,t,o,i\rangle] \\
\mathbf{0} &\stackrel{\text{def}}{=} P_{(p,t,o,i)}[\, p|t|o|i\,] \\
\langle u\rangle.Q &\stackrel{\text{def}}{=} P_{(p,t,o,i)}[\,(\nu q)(\mathsf{out}(p,q,u,o)|Q\langle q,t,o,i\rangle)\,] \\
\langle u\rangle^{\uparrow}.P &\stackrel{\text{def}}{=} P_{(p,t,o,i)}[\,(\nu q)(\mathsf{ret}(p,q,u,t)|Q\langle q,t,o,i\rangle)\,] \\
(u).P &\stackrel{\text{def}}{=} P_{(p,t,o,i)}[\,(\nu q)(\mathsf{in}(p,q,u,i)|Q\langle q,t,o,i\rangle)\,]
\end{aligned}
$$

Part of the above definition is graphically represented in Fig. 4. As in Section 3 we use different arrow types to denote the different (ordered, typed) tentacles of each edge. For example, for a design representing a process, a double arrow represents its returning channel, an outgoing arrow its output channel, an incoming arrow its input channel and a plain arrow its control point. Again, arguments of an operation are denoted by encircling the corresponding symbol. For instance, double boxes correspond to design variables, while node arguments of type $\circ$, $\star$ and $\square$ are represented by $\circledcirc$, $\circledast$ and $\boxdot$, respectively.

We introduce flattening axioms $\mathsf{flat_P}$ into $\equiv_d$, but not $\mathsf{flat_S}$, $\mathsf{flat_D}$, $\mathsf{flat_I}$, $\mathsf{flat_F}$ and $\mathsf{flat^N_T}$. Hence, edges of type P are immaterial (they can be considered as type annotations) and the only explicit hierarchies are given by session nesting (S), service definition (D), service invocation (I) and pipelining (F and T). Flattening processes allows for getting rid of the axioms for parallel composition (see [1]). The explicit embedding of sessions provides an intuitive visual representation.

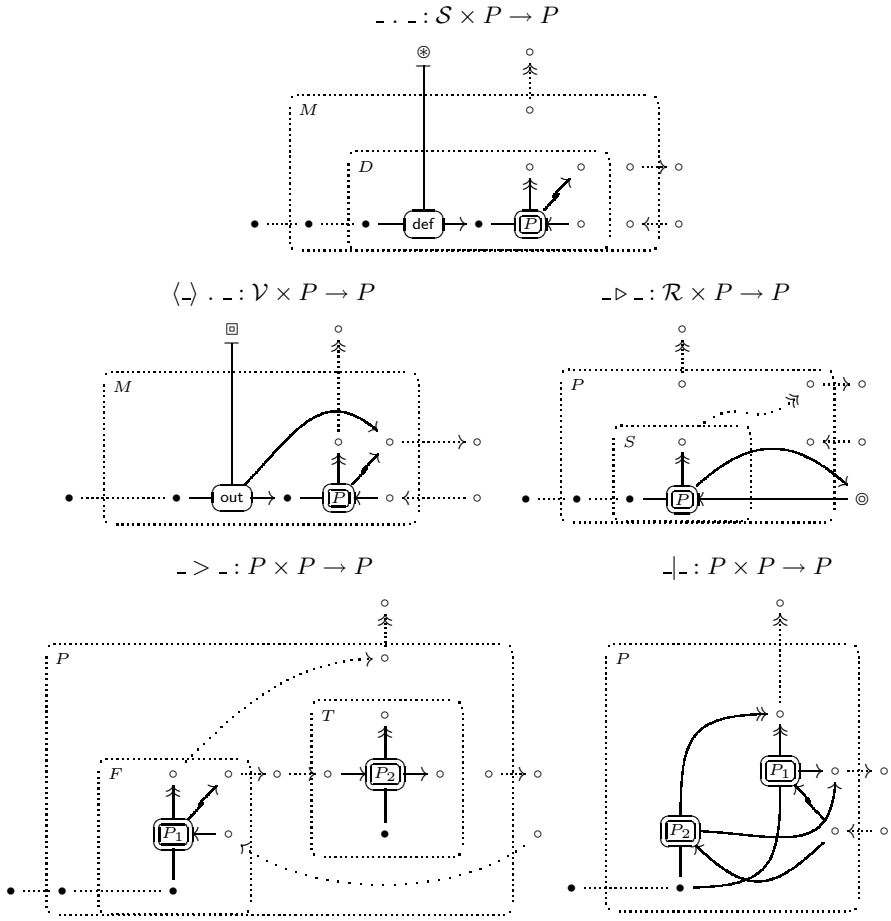

Fig. 4. Graphical representation of some CaSPiS interpreted operators.

We explain just a few representative operations in detail. The session operations are interpreted as graph operations that wrap a process into a hierarchical S-typed graph which exposes the control point and a return channel. The first is associated to the control point of the resulting P-typed design, while the second is connected to its output channel. Note how session embedding hides the input and output channels of the embedded process: they are connected directly to the dedicated inter-communication node of the session. Another interesting operation is the pipeline. Here, the source and target process of the pipeline are embedded in F- and T-typed designs. It is worth noting how the input and output channels of each process are connected in a complementary way. The target process hides its control point and communication channels to denote that it is a non-active process. When the source of the pipe is ready to send a value, a copy of the target

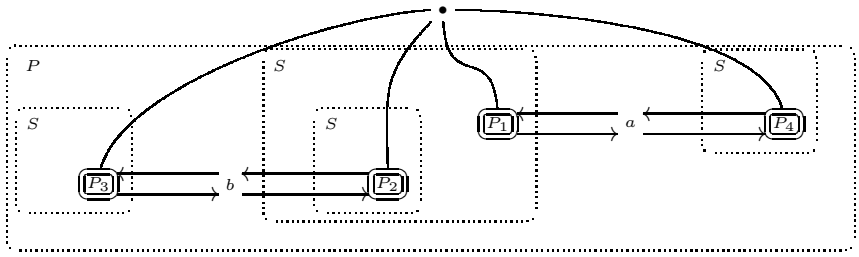

Fig. 5. Example of session nesting.

process will be created and the control and channel nodes will be connected as expected. Moreover, we note that the actual type for the target of the pipe is $T^{fn(R)}$: in words, the type is indexed with the free names of R. This is necessary to avoid node extrusion in a case in which we have no corresponding name extrusion (CaSPiS congruence does not allow to extrude restricted names of the target process of a type). In particular when $w \in fn(R)$ the CaSPiS processes $(\nu w)(Q > R)$ and $Q > (\nu w)R$ are not congruent, but neither are their corresponding graphs $P_{(p,t,i,o)}[(\nu q, m, w)(F_{\langle p,t,m,i\rangle}[Q\langle p,t,m,i\rangle]|T^{fn(R)}_{\langle m\rangle}[(\nu q,t',i')R\langle q,t',o',m\rangle])]$ and $P_{(p,t,i,o)}[(\nu q, m)(F_{\langle p,t,m,i\rangle}[Q\langle p,t,m,i\rangle]|T^{fn(R)\setminus\{w\}}_{\langle m\rangle}[(\nu q,t',i',w)R\langle q,t',o',m\rangle])]$, because they carry different T subtypes.

Example 3. Let us illustrate our encoding with a simple example of session nesting. Consider process $(\nu a)(\nu b)(a \triangleright (P_1|b \triangleright P_2)|a \triangleright P_3|b \triangleright P_4)$. Two sessions a and b have been created (as the result of two service invocations). Agent $a \triangleright (P_1|b \triangleright P_2)$ participates to sessions a and b (assume P_1 is the protocol for a and P_2 the one for b), with the b side nested in a. The counter-party protocols for a and b are P_3 and P_4, respectively. Figure 5 depicts the graphical representation of our example, where the graph has been simplified[3] (e.g. fusing nodes, removing isolated nodes and irrelevant tentacles) to focus on the main issues.

Example 4. As another illustrative example consider processes $P_1 > (P_2 > P_3)$ whose (simplified) graphical representation is in Fig. 6. The graphical representation highlights various aspects of interest: the flow of the information via the input and output channels, the fact that P_2 and P_3 are *inactive* protocols, and the pipe nesting; since $>$ is not associative $P_1 > (P_2 > P_3)$ and $(P_1 > P_2) > P_3$ are not structurally equivalent and this is faithfully reflected in the graphs.

Full-detailed examples can be obtained via our prototypical implementation.

A main result of our work is that structural congruence amounts to design equivalence, i.e. equivalent processes amount to equivalent graphs.

Proposition 2. *For any two processes P and Q we have $P \equiv_c Q$ iff $P \equiv_d Q$.*

[3] A feature that a sophisticated visualisation tool would offer.

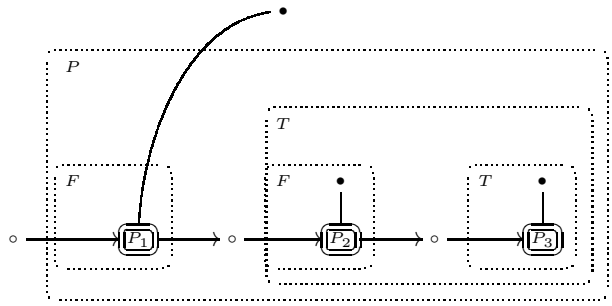

Fig. 6. Example of pipelining.

5 Conclusion

We have presented a first step towards a general technique for the graphical presentation of (possibly service-oriented) process calculi.

More precisely, we have used our novel specification formalism based on a convenient algebra of hierarchical graphs [7] to define encodings of process calculi with inherently hierarchical aspects such as sessions, transactions or locations: features which are of fundamental relevance, e.g. in the area of service-oriented computing. In particular, together with the encoding of a simple language for structured workflow with nest scope, we have presented a novel graphical encoding of CaSPiS, a recently proposed session-centered calculus. Our document also included an encoding of a simple workflow language.

The chosen encodings highlight the virtues of our graph algebra. First, its syntax resembles the standard syntax of a process calculus, thus offering the possibility of providing intuitive and simple encoding definitions. Second, we can exploit the algebraic structure of both processes and graphs to show encoding properties by structural induction. Indeed the main result of [7] already guarantees that equivalent graph terms correspond to isomorphic graphs.

As explained in [7], the particular model of hierarchical graphs puts on a common ground other approaches that have been issued for modelling purposes like the algebra of graphs of Corradini et al. [9], the interface graphs of the second author [1] (a flat model for encoding process calculi with names), the hierarchical graphs of Plump et al. [11] (a suitable extension of traditional graph transformation) and the Bigraphs of Milner et al. [2]. We refer to [7] for a more accurate comparison and we only remark here that one of the advantages of our model of graphs regards the use of hierarchical edges over trees to model structured processes. In unstructured cases both approaches are basically equivalent, with hierarchies possibly offering a more attractive visualisation. However, when it comes to structured process calculus where one has recursive processes (either in the form of process definitions, replication, pipes, etc.), we observe a major advantage in the use of hierarchical graphs, where the replication of the structured process is easier.

Our final goal is to completely mimick the operational semantics of encoded processes and in this line we believe that our model enjoys some good properties, the main being that (though not shown here) the category of hierarchical graphs is complete (puhshouts are well defined) which puts the basis for a pushout-based graph rewriting mechanism. While the mimicking of reduction semantics seems rather straightforward we also point to the more ambitious goal to mimick labelled transition system semantics, possibly in the form of SOS rules. For that purpose we expect that recent approaches based on borrowed contexts [12] or structured graph transformation [13] can be a good start point. The development of a suitable dynamics for our algebras, and its characterisation in terms of graph rewriting mechanisms, is subject of current work.

We believe that our approach can serve as an inspiration to equip well-known graphical models of communication with syntactical notations that facilitate the definition of intuitive and correct encodings of process calculi. We remark that we restricted our attention to the finite fragment of the calculi. Nevertheless, dealing with replication operators is by no means difficult, by exploiting the hierarchical structure. Of course, the axiom $!P \equiv !P \mid P$ would not hold, since the two terms would have different graphical encoding. However, it would suffice to introduce an unfolding operation, possibly parametric in the free names of P, as it happens for the encoding of pipe operators in CaSPiS.

Even if not presented here, we have also applied our technique to other calculi. For instance, in [7] we offer an encoding of the best-known nominal calculus, the π-calculus. The encoding is roughly equivalent to the one in [1], and interested readers are invited to compare the two proposals to get a clear idea of the convenience of using our graph algebra in the definition of the encoding and, most importatly, on its proof of correctness. We have also focused on service oriented calculi testing our technique on a calculus of transactions called *sagas* [14], and a calculus with locations and multi-party sessions called μse [15].

An implementation of the presented approach and its integration in our prototypical implementation of ADR [16] is currently under development. A preliminary version is already available[4] in the form of a visualiser. We remark that our approach proposes a *graphical* representation of configurations but it is not intended to be a *visual* representation: the graph structures we propose might be used as the formal and facilitating support of some particular kind of appealing diagrams. This issue remains to be investigated.

References

1. Gadducci, F.: Term graph rewriting for the pi-calculus. In Ohori, A., ed.: APLAS 2003. Volume 2895 of Lecture Notes in Computer Science., Springer (2003) 37–54
2. Jensen, O.H., Milner, R.: Bigraphs and mobile processes. Technical Report 570, Computer Laboratory, University of Cambridge (2003)
3. Ferrari, G.L., Hirsch, D., Lanese, I., Montanari, U., Tuosto, E.: Synchronised hyperedge replacement as a model for service oriented computing. In de Boer,

[4] www.albertolluch.com/adr2graphs/

F.S., Bonsangue, M.M., Graf, S., de Roever, W.P., eds.: FMCO 2005. Volume 4111 of Lecture Notes in Computer Science., Springer (2006) 22–43
4. Gadducci, F., Monreale, G.V.: A decentralized implementation of mobile ambients. In Ehrig, H., Heckel, R., Rozenberg, G., Taentzer, G., eds.: ICGT 2008. Volume 5214 of Lecture Notes in Computer Science., Springer (2008) 115–130
5. Bundgaard, M., Sassone, V.: Typed polyadic pi-calculus in bigraphs. In Bossi, A., Maher, M.J., eds.: PPDP 2006, ACM (2006) 1–12
6. Grohmann, D., Miculan, M.: An algebra for directed bigraphs. In Mackie, I., Plump, D., eds.: TERMGRAPH 2007. Volume 203 (1) of Electronic Notes in Theoretical Computuer Science., Elsevier (2008) 49–63
7. Bruni, R., Gadducci, F., Lluch Lafuente, A.: An algebra of hierarchical graphs. Submitted. Available at http://www.albertolluch.com/papers/adr.subitted.pdf (2009)
8. Boreale, M., Bruni, R., Nicola, R.D., Loreti, M.: Sessions and pipelines for structured service programming. In Barthe, G., de Boer, F.S., eds.: FMOODS 2008. Volume 5051 of Lecture Notes in Computer Science., Springer (2008)
9. Corradini, A., Montanari, U., Rossi, F.: An abstract machine for concurrent modular systems: CHARM. Theoretical Compututer Science **122** (1994) 165–200
10. Drewes, F., Kreowski, H.J., Habel, A.: Hyperedge replacement, graph grammars. In Rozenberg, G., ed.: Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations, World Scientific (1997) 95–162
11. Drewes, F., Hoffmann, B., Plump, D.: Hierarchical graph transformation. Journal on Computer and System Sciences **64** (2002) 249–283
12. Bonchi, F., Gadducci, F., Monreale, G.V.: Reactive systems, barbed semantics, and the mobile ambients. In de Alfaro, L., ed.: FOSSACS 2009. Volume 5504 of Lecture Notes in Computer Science., Springer (2009) 272–287
13. Drewes, F., Hoffmann, B., Janssens, D., Minas, M., Eetvelde, N.V.: Shaped generic graph transformation. In Schürr, A., Nagl, M., Zündorf, A., eds.: AGTIVE 2007. Volume 5088 of Lecture Notes in Computer Science., Springer (2007) 201–216
14. Bruni, R., Melgratti, H.C., Montanari, U.: Theoretical foundations for compensations in flow composition languages. In Palsberg, J., Abadi, M., eds.: POPL 2005, ACM (2005) 209–220
15. Bruni, R., Lanese, I., Melgratti, H.C., Tuosto, E.: Multiparty sessions in SOC. In Lea, D., Zavattaro, G., eds.: COORDINATION 2008. Volume 5052 of Lecture Notes in Computer Science., Springer (2008) 67–82
16. Bruni, R., Lluch Lafuente, A., Montanari, U.: Hierarchical design rewriting with maude. In Rosu, G., ed.: WRLA 2008. Volume 238 (3) of Electronic Notes in Theoretical Computer Science., Elsevier (2009) 45–62