

# Hierarchical Graphs for Service-oriented Calculi<sup>\*</sup>

Roberto Bruni, Fabio Gadducci, and Alberto Lluch Lafuente

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

**Abstract.** We introduce a novel specification formalism based on hierarchical graphs. More precisely, we define a term-like syntax for describing a class of hierarchical graphs, and we present an equational characterization for graph isomorphism, i.e. a sound and complete set of axioms equating two terms whenever they represent the same hierarchical graph. Our equational presentation can thus be understood as a high-level language for describing graphs with a nested structure, and is then particularly suited to define graphical encodings of service oriented calculi, which typically exhibit hierarchical features such as sessions and transactions. We illustrate our approach by encoding a session-centered calculus.

## 1 Introduction

Graphs are suitable formalisms to represent software systems as exemplified by the many graph-based approaches proposed for systems with features such as distribution, concurrency and mobility. We cite, among others, those based on traditional graph transformation [9], Bigraphs [11] and Synchronized Hyperedge Replacement [8]. Using such approaches to build a graphical representation of an existing language involves two major challenges: encoding states and encoding the operational semantics. When the language is defined as a process calculus, for instance, the encoding of a state is facilitated by the algebraic structure of states (i.e. as terms) and 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 such sessions or transactions. As a result, typical solutions require advanced skills to the encoding designers and are based on sophisticated techniques involving graphs with interfaces (e.g. [9]), enriched type system (e.g. [10]) or hierarchies as trees (e.g. [11]) which further complicate the proof of correctness of the encoding.

We introduce a novel specification formalism made of a term-like syntax for describing a class of hierarchical graphs equipped with an equational characterization for graph isomorphism, that is, 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, restriction, parallel composition and nesting, particularly suited to define graphical encodings of service oriented calculi which typically exhibit hierarchical features such as sessions and transactions.

---

<sup>\*</sup> Supported by the EU-GC2 FETPI project IST-2005-016004 SENSORIA.

We validate our idea by encoding two calculi. The first one is a well-known calculus, namely the  $\pi$ -calculus, for which various graphical encodings exist and that has no hierarchical features. The second one is a calculus for the description of service oriented applications, namely CaSPiS [1], which has a hierarchical feature (sessions) and that has been never encoded into graphs before. Though our focus is put on a particular process calculus, we remark that our technique can be applied to other calculi.

*Structure of the Paper.* Section 2 presents our basic algebra of hierarchical graphs. The model of graphs underlying such algebra is described in Section 3. Sections 4 and 5 present our graphical interpretation of the  $\pi$ -calculus and CaSPiS. Section 6 discusses related work. Section 7 concludes our paper and outlines future research avenues. An appendix is included for the convenience of reviewers.

## 2 An algebra of hierarchical graphs

We present here for the first time our algebra of typed, hierarchical graphs that we call *designs*. The algebraic presentation of designs is inspired by *Architectural Design Rewriting* [3] (hence the name *design*) and the graph algebra of [4].

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

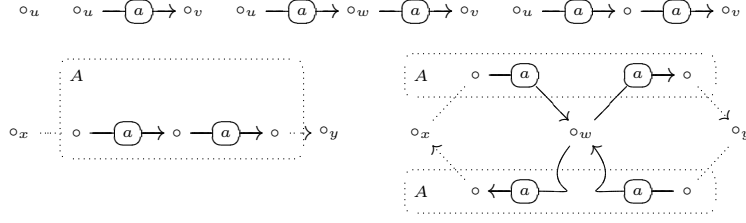
$$\mathbb{D} ::= L_{\bar{x}}[\mathbb{G}] \quad \mathbb{G} ::= \mathbf{0} \mid x \mid l(\bar{x}) \mid \mathbb{G} \mid \mathbb{G} \mid (\nu x)\mathbb{G} \mid \mathbb{D}(\bar{x})$$

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  $\bar{x} \in \mathcal{N}^*$ .

Terms generated by  $\mathbb{G}$  are (possibly hierarchical) graphs. More precisely,  $\mathbf{0}$  represents the flat empty graph,  $x$  is a discrete graph containing node  $x$  only,  $l(\bar{x})$  is a graph formed by an  $l$ -labelled edge attached to nodes  $\bar{x}$  (the  $i$ -th tentacle to the  $i$ -th node in  $\bar{x}$ ),  $\mathbb{G} \mid \mathbb{H}$  is the graph resulting from the parallel composition of graphs  $\mathbb{G}$  and  $\mathbb{H}$  (the disjoint union up to common nodes),  $(\nu x)\mathbb{G}$  is graph  $\mathbb{G}$  after restricting node  $x$ , and  $\mathbb{D}(\bar{x})$  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  $L_{\bar{x}}[\mathbb{G}]$  represents a design of type  $L$ , with body graph  $\mathbb{G}$  and exposing nodes  $\bar{x}$  in its interface. We remark that the interface  $\bar{x}$  is a vector and not a set. As a consequence a node can appear twice, thus allowing for a design to expose an internal node more than once. For instance,  $L_{(x,x)}[x]$  is a design with one node  $x$  being exposed twice in the interface.

*Example 1.* Let  $a \in \mathcal{T}$ ,  $A \in \mathcal{NT}$ ,  $u, v, w, x, y \in \mathcal{N}$ . We write and depict (Figure 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  $A[\lambda(u, v).(\nu w)(a(u, w) \mid a(w, v))](x, y)$  (bottom-left). Nodes are represented by circles, edges by boxes, 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



**Fig. 1.** Some terms of the graph algebra.

exposure of nodes in the interface of a design is denoted by dotted arrows. Nodes subscripted with their identities are nodes that are neither being exposed nor restricted. The rest are interface or restricted nodes. The picture also includes term  $A_{(u,v)}[a(u,w) \mid a(w,v)](x,y) \mid A_{(u,v)}[a(u,w) \mid a(w,v)](y,x)$  (bottom-right), where two designs are composed in circle by attaching them symmetrically to  $x$  and  $y$ , and share (as a common name) node  $w$ .

The main difference with respect to the algebra of [4] is the introduction of hierarchical graphs and the presence of isolated nodes: a term  $x \mid \mathbb{G}$  stands for a graph where node  $x$  is isolated whenever not attached to any edge in  $\mathbb{G}$ .

Sort  $\mathbb{D}$  is partitioned over the  $\mathcal{NT}$ , i.e. we consider sorts  $L_1, \dots, L_n$  (where  $\mathcal{NT} = \{L_1, \dots, L_n\}$ ) and a membership  $\mathbb{D} : L$  whenever  $\mathbb{D} = L_{\bar{x}}[\mathbb{G}]$ . 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. We will see that this useful to distinguish nodes representing communication channels, values and service names.

The presence of node restriction and interface 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 function  $fn$  defined as follows:*

$$\begin{aligned} fn(\mathbf{0}) &= \emptyset & fn(x) &= x & fn(l(\bar{x})) &= |\bar{x}| & fn(\mathbb{G} \mid \mathbb{H}) &= fn(\mathbb{G}) \cup fn(\mathbb{H}) \\ fn(L_{\bar{x}}[\mathbb{G}]) &= fn(\mathbb{G}) \setminus |\bar{x}| & fn((\nu x)\mathbb{G}) &= fn(\mathbb{G}) \setminus \{x\} & fn(\mathbb{D}(\bar{x})) &= fn(\mathbb{D}) \cup |\bar{x}| \end{aligned}$$

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

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 typed tentacles of hyper-edges with that label. The typed arities of a label  $l$ , node vector  $\bar{x}$  and design  $\mathbb{D}$ , are respectively denoted by  $t(l)$ ,  $t(\bar{x})$ , and  $t(\mathbb{D})$ , where the latter is an abbreviation for  $t(L)$ , with  $\mathbb{D} : L$ . We require the types to be respected and nodes being exposed in a design to be actually free in the body graph.

**Definition 3 (well-formedness).** *A design or graph is well-formed whenever for each occurrence of  $L_{\bar{x}}[\mathbb{G}]$  we have  $\bar{x} \subseteq fn(\mathbb{G})$  and  $t(\bar{x}) = t(L)$ , for each occurrence of  $\mathbb{D}(\bar{y})$  we have  $t(\mathbb{D}) = t(\bar{y})$  and for each occurrence of  $l(\bar{x})$  we have  $t(\bar{x}) = t(l)$ .*

From now on we restrict to well-formed designs: axioms will preserve well-formedness and all derived operators will be well-formed.

In order to have a notion of syntactically equivalent designs (i.e. to consider designs up to isomorphism) we introduce the usual structural graph axioms [4] in the algebra such as AC1 for  $|$  (with identity  $\mathbf{0}$ ) and name extrusion. In addition, we include axioms to  $\alpha$ -rename interface nodes, node extrusion for designs, and an axiom to consider immaterial the addition of a node in parallel with a graph for which the node is already free.

**Definition 4 (design axioms).** *Structural congruence  $\equiv_d$  of designs and graphs is given by the least congruence satisfying:*

$$\begin{array}{ll}
\mathbb{G} \mid \mathbb{H} \equiv \mathbb{H} \mid \mathbb{G} & (\nu x)\mathbb{G} \equiv (\nu y)\mathbb{G}\{y/x\} \quad \text{if } y \notin \text{fn}(\mathbb{G}) \\
\mathbb{G} \mid (\mathbb{H} \mid \mathbb{I}) \equiv (\mathbb{G} \mid \mathbb{H}) \mid \mathbb{I} & L_{\bar{x}}[\mathbb{G}] \equiv L_{\bar{y}}[\mathbb{G}\{y/\bar{x}\}] \quad \text{if } |\bar{y}| \cap \text{fn}(\mathbb{G}) = \emptyset \\
\mathbb{G} \mid \mathbf{0} \equiv \mathbb{G} & \mathbb{G} \mid (\nu x)\mathbb{H}(\nu x) \equiv (\mathbb{G} \mid \mathbb{H}) \quad \text{if } x \notin \text{fn}(\mathbb{G}) \\
(\nu x)(\nu y)\mathbb{G} \equiv (\nu y)(\nu x)\mathbb{G} & L_{\bar{x}}[(\nu y)\mathbb{G}](\bar{z}) \equiv (\nu y)L_{\bar{x}}[\mathbb{G}](\bar{z}) \quad \text{if } y \notin |\bar{x}| \cup |\bar{z}| \\
(\nu x)\mathbf{0} \equiv \mathbf{0} & x \mid \mathbb{G} \equiv \mathbb{G} \quad \text{if } x \in \text{fn}(\mathbb{G})
\end{array}$$

where in the  $\alpha$ -renaming axiom for interface nodes we require the substitution  $\{y/\bar{x}\}$  to be a bijection to avoid node fusion.

We call a design *flat* whenever there is no design in its body. Flattening a design can be done by a kind of hyper-edge replacement [7]. We define such flattening as axioms that we might include them in the structural congruence when useful.

**Definition 5 (flattening axiom).** *A flattening axiom  $\text{flat}_L$  is of the form  $L_{\bar{x}}[\mathbb{G}](\bar{y}) \equiv \mathbb{G}\{y/\bar{x}\}$ , for some design label  $L$ .*

*Example 2.* Suppose that we want to characterise the set of  $a$ -labelled acyclic, connected sequences (see our previous example). We can define an algebra with the empty sequence  $\epsilon$ , an element  $\alpha$  in the sequence, and a binary sequential composition  $;$ ;  $\_$ . All are derived operators defined by  $\epsilon \stackrel{\text{def}}{=} A_{(u,u)}[u]$ ,  $\alpha \stackrel{\text{def}}{=} A_{(u,v)}[a(u,v)]$  and  $X;Y \stackrel{\text{def}}{=} A_{(u,v)}[(\nu w)X(u,w) \mid Y(w,v)]$ . Clearly, the algebra as such constructs hierarchical sequences with  $;$ , where  $\alpha;(\alpha;\alpha)$  and  $(\alpha;\alpha); \alpha$  are not equivalent graphs. Introducing  $\text{flat}_A$  in the algebra, instead, we have that the former terms are both equivalent to  $(\nu w_1, w_2)(a(w, w_1) \mid a(w_1, w_2) \mid a(w_2, v))$ .

### 3 A model of hierarchical graphs

This section introduces the model for the algebra we described in the previous section. We draw our inspiration from the hierarchical graphs defined in [6], which we equip with a suitable notion of interface.

**Definition 6 (hypergraph).** *A hypergraph  $G$  is a triple  $\langle E_G, N_G, t_G \rangle$  such that  $E_G$  ( $N_G$ ) is the set of edges (nodes), and  $t_G : E_G \rightarrow N_G^*$  is the tentacle function.*

*Let  $G, H$  be hypergraphs. A (hypergraph) morphism  $f : G \rightarrow H$  is a pair of functions  $f_E : E_G \rightarrow E_H$ ,  $f_N : N_G \rightarrow N_H$  preserving the tentacle function.*

In the definition above we consider sets of nodes and edges to be drawn from the (sorted) universes defined in the previous section. In the following, we indifferently use  $\mathcal{G}$  to denote either the category of hypergraphs or the set of all hypergraphs. We now introduce a refinement on the notion of isomorphism.

**Definition 7 (isomorphisms up-to).** *Let  $\sigma$  be an endofunctor on  $\mathcal{G}$ . Two hypergraphs  $G, H$  are isomorphic up-to  $\sigma$  if  $\sigma(G)$  is isomorphic to  $\sigma(H)$ .*

In the following, we often consider the functor taking away (part of) the isolated nodes of an hypergraph. Now we recall the definition of hierarchical graph offered in [6], adapting it to our purposes.

**Definition 8 (hierarchical hypergraphs).** *The family  $\mathcal{H}$  of hierarchical hypergraphs (HHS) is the union of all families  $\mathcal{H}_i$  of level  $i$ , with  $i \in \mathbb{N}$ , where  $\mathcal{H}_0 = \mathcal{G}$  and  $\mathcal{H}_{i+1} = \{\langle G, I \rangle \mid G \in \mathcal{G} \wedge I : E_G \rightarrow H_i\}$ .*

In words,  $\mathcal{H}_0$  is the set of all (flat) hypergraphs,  $\mathcal{H}_1$  is the set of all hierarchical graphs of level 1, meaning that some edges are mapped to flat graphs, and so on. The set  $\mathcal{H}_{i+1}$  is the set of all hierarchical hypergraphs of level  $i+1$  formed by a graph and a partial mapping assigning a graph of level  $i$  to some edges.

Since functions  $I$  are partial, each level  $\mathcal{H}_i$  can be embedded into  $\mathcal{H}_{i+1}$  in a simple way, by using a suitable coercion function  $\rho$  inductively defined as  $\rho_0(G) = \langle G, \perp \rangle$  and  $\rho_{i+1}(\langle G, I \rangle) = \langle G, I \circ \rho_i \rangle$ . The symbol  $\perp$  denotes the always empty function. Thus, we can abuse notation by saying that  $\mathcal{H}_i \subseteq \mathcal{H}_{i+1}$ .

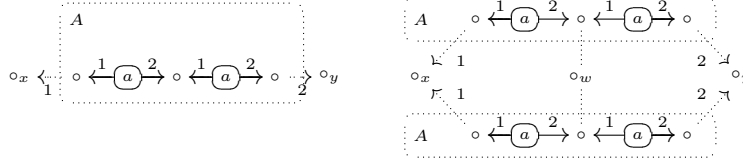
HHS are purely hierarchical in the sense that nodes cannot be shared between hierarchical levels. We now extend them to allow the modelling of node sharing, as allowed in our algebra of designs.

**Definition 9 (extended hierarchical hypergraphs).** *The family  $\mathcal{E}$  of extended hierarchical hypergraphs (EHS) is inductively defined as  $\mathcal{E}_0 = \mathcal{G}$  and  $\mathcal{E}_{i+1} = \{\langle G, I, X \rangle \mid G \in \mathcal{G} \wedge I : E_G \rightarrow H_i \wedge \forall e \in E_G.X_e : N_G \rightarrow N_{I(e)}\}$  such that  $\forall e \in E_G.X_e$  is total over  $t_G(e)$ .*

In other terms,  $X = \{X_e \mid e \in E_G\}$  is a family of partial functions associating nodes in  $G$  to nodes in the hypergraph occurring in the EHS  $I(e)$ , denoted by an abuse of notation as  $N_{I(e)}$ . Thus, a node in a hypergraph  $\langle G, I, X \rangle \in \mathcal{E}_{i+1}$  can be mapped to a set of nodes occurring in hypergraphs belonging to  $\mathcal{E}_i$ . In particular, all the nodes belonging to  $t_G(e)$  must have an image in  $I(e)$ . Intuitively,  $X_e$  can be seen as mapping nodes to the corresponding local copies in the graph embedded in  $e$ .

**Definition 10 (hierarchical morphisms).** *Let  $\langle G, I \rangle, \langle H, J \rangle$  be HHS in  $\mathcal{H}_i$ . A (HH) morphism (of level  $i$ )  $\langle f, f_e \rangle : \langle G, I \rangle \rightarrow \langle H, J \rangle$  is a morphism  $f : G \rightarrow H$  plus a family of morphisms (of level  $i-1$ )  $f_e : I(e) \rightarrow J(f(e))$  for each  $e \in E_G$ .*

*Let  $\langle G, I, X \rangle, \langle H, J, Y \rangle$  be EHS in  $\mathcal{E}_i$ . A (EHS) morphism (of level  $i$ )  $\langle f, f_e \rangle : \langle G, I, X \rangle \rightarrow \langle H, J, Y \rangle$  is a HH morphism (of level  $i$ ) such that  $f_e(X_e(n)) = Y_{f(e)}f(n)$  for each  $n \in N_G, e \in E_G$ .*



**Fig. 2.** Some terms of the graph algebra interpreted as hierarchical graphs.

Morphisms up-to are obviously defined, once an endofunctor on  $\mathcal{G}$  is chosen.

Let  $\langle G, I, X \rangle$  be a hypergraph and  $n \in N_G$  a node. We say that a node  $n$  is *isolated* if there is no edge  $e \in E_G$  such that  $n \in t_G(e)$ ; *detached* if there is no edge  $e \in E_G$  such that  $X_e(n)$  is defined; and *localized* if there is only one edge  $e \in E_G$  such that  $X_e(n)$  is defined.

Note that an isolated node might not be detached: that would be the case if the node is being used in lower levels. As a matter of fact being detached implies being also isolated. A node is *detachable* if it is detached (no one is using it) or it is both localized and isolated (used in the underlying graph of a single edge only). We shall consider hypergraphs up-to detachable nodes.

The final step is to consider a hypergraph with names, by equipping EHHs with a (disjoint) declaration of free and parameter nodes.

**Definition 11.** An extended hierarchical hypergraph with interfaces (EHHWI) is a 5-tuple  $\langle G, I, X, F, V \rangle$  such that  $\langle G, I, X \rangle$  is an EHH,  $F \subseteq N_G$  is the subset of free nodes and  $V \in N_G^*$  is the tuple of parameter nodes, such that  $F \cap |V| = \emptyset$ .

Let  $\langle G, I, X, F, V \rangle, \langle H, J, Y, F, W \rangle$  be EHHWIs. A (EHHWI) morphism  $\langle f, f_e \rangle : \langle G, I, X, F, V \rangle \rightarrow \langle H, J, Y, F, W \rangle$  is a EHH morphism such that  $f(n) = n$  for each  $n \in F$ .

In other terms, we assume that EHHWI morphisms only relate hypergraphs with the same set (not just up-to isomorphism) of free nodes. The notion of isomorphism up-to is obviously defined: in the following, we will consider the functor dropping at each level those hidden (i.e. not free) nodes that are also detachable.

*Interpreting designs as EHHWIs.* We now see how terms of our algebra of designs can be interpreted as EHHWIs.

**Definition 12 (Design interpretation).**

$$\begin{aligned}
[[x]] &= \langle \langle x, \emptyset, \emptyset \rangle, \emptyset, \emptyset, \emptyset, \emptyset \rangle & [[l(\bar{x})]] &= \langle \langle |x|, e, e \mapsto \bar{x} \rangle, \perp, \emptyset, |x|, \emptyset \rangle \\
[[(\nu x)\mathbb{G}]] &= \langle G_{\mathbb{G}}, I_{\mathbb{G}}, X_{\mathbb{G}}, F_{\mathbb{G}} \setminus x, \emptyset \rangle & [[\mathbf{0}]] &= \langle \emptyset, \emptyset, \emptyset, \emptyset, \emptyset \rangle \\
[[\mathbb{G} \mid \mathbb{H}]] &= \langle G_{\mathbb{G}} \uplus H_{\mathbb{H}}, I_{\mathbb{G}} \uplus I_{\mathbb{H}}, X_{\mathbb{G}} \uplus X_{\mathbb{H}}, F_{\mathbb{G}} \cup F_{\mathbb{H}}, \emptyset \rangle \\
[[\mathbb{D}(\bar{x})]] &= \langle G_{\mathbb{D}}, I_{\mathbb{D}}, X_{\mathbb{D}}, F_{\mathbb{D}} \cup |\bar{x}|, \emptyset \rangle & & \text{if } \mathbb{D} : L \text{ and } \text{flat}_{\mathbb{L}} \not\equiv_d \\
[[\mathbb{D}(\bar{x})]] &= \langle I_{\mathbb{D}}(e) \{ \bar{x} / F_{I_{\mathbb{D}}(e)} \}, I_{I_{\mathbb{D}}(e)}, X_{I_{\mathbb{D}}(e)}, F_{I_{\mathbb{D}}(e)} \cup |\bar{x}|, \emptyset \rangle & & \text{if } \mathbb{D} : L \text{ and } \text{flat}_{\mathbb{L}} \equiv_d \\
[[L_{\bar{x}}[\mathbb{G}]]] &= \langle e' \cup F, e' \mapsto \langle G_{\mathbb{G}}, I_{\mathbb{G}}, X_{\mathbb{G}} \rangle, e' \mapsto id_F, F, \bar{x} \rangle
\end{aligned}$$

where  $e : l, e' : L$ .

Abusing notation, in the definition above  $\emptyset$  denotes empty sets, vectors and mappings. We denote by  $\uplus$  the obvious disjoint union of graphs (up to free nodes). This involves renaming graph items, which can be freely chosen because we are interested in graphs up-to isomorphism: the only requirement is the merging of common free nodes. We also assume that subscripts refer to the corresponding encoded graph. For instance,  $\llbracket G \rrbracket = \langle G_{\mathbb{G}}, I_{\mathbb{G}}, X_{\mathbb{G}}, F_{\mathbb{G}}, V_{\mathbb{G}} \rangle$ . Note that interpretation of  $\mathbb{D}(\bar{x})$  depends on the presence of flattening axioms for the type of  $\mathbb{D}$ .

The encoding of the empty graph, an isolated node and a single edge are easy to understand. The encoding of the parallel composition is also as expected: it is basically a disjoint union of the corresponding hierarchical graphs. Node restriction consists of removing the node from the set of free nodes. A hierarchical edge is basically encoded as a graph containing a single edge (which is mapped to the corresponding body graph) and a copy of the free nodes of the body graph (properly mapped to the corresponding copies in the body). Finally, the encoding of  $D(\bar{x})$  is obtained simply by making empty the tuple of parameter nodes, and adding the names among the free ones.

It is worth noting that the axioms of our graph algebra amount to our notion of graph isomorphism. For example, the encoding of  $(\nu x)\mathbf{0}$  is the same as the encoding of  $\mathbf{0}$ , up-to removing the hidden (i.e. not free) detached node representing the name  $x$ . Also the encoding of  $L_{\bar{x}}[(\nu y)G](\bar{z})$  is the same of the encoding of  $(\nu y)(L_{\bar{x}}[G](\bar{z}))$ , assuming  $y \notin \bar{x} \cup \bar{z}$ , since in the latter case the node representing  $y$  is both hidden and detachable. So, the former is a normal form of the latter, with the detachable node at the higher level removed.

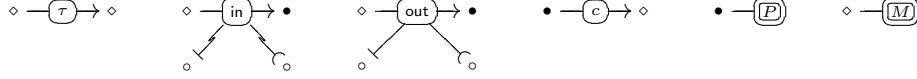
We denote say that two graphs  $G, H$  are *equivalent* (denoted  $G \equiv_g H$  if they are isomorphic up-to the removal of hidden, detachable nodes.

**Theorem 1.** *Let  $t_1, t_2$  be terms (graph or designs) generated by the design algebra. Then,  $t_1 \equiv_a t_2$  if and only if  $\llbracket t_1 \rrbracket \equiv_g \llbracket t_2 \rrbracket$ .*

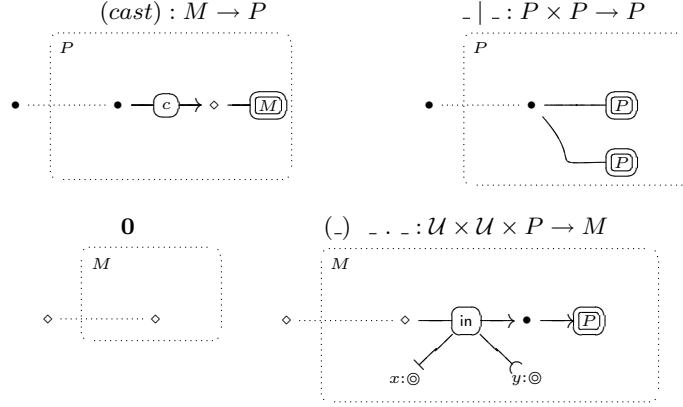
*Example 3.* Consider the graphs and terms of Example 1, visualized in Figure 1. The actual hierarchical graphs interpretation (for the non-flat graphs) is instead depicted in Figure 2. Note that the informal representation of terms of our graph algebra is similar to that of hierarchical graphs but includes convenient shorthands such as node-sharing represented by a unique nodes and tentacles crossing the hierarchy levels. Furthermore, the ordering on the tentacles out of each edge is dropped, and an informal notation, often removing heads or adding tails to tentacles, is chosen whenever it seems suggestive for the system we are modeling. In the rest of our paper we continue with the informal representation for its visual appealing, but it should be clear that there is a formally define underlying hypergraph.

## 4 Graphs for the $\pi$ -calculus

This section aims at offering a first intuition our approach by presenting, in an illustrative way, an encoding of the well-known  $\pi$ -calculus [12]. A minimal



**Fig. 3.** Node and edge sorts.



**Fig. 4.** Excerpt of the graphical encoding of the  $\pi$ -calculus.

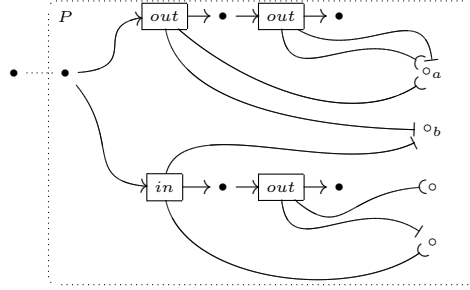
familiarity with the calculus can be helpful but our presentation here offers sufficient hints for our purpose.

We start presenting (cf. Figure 3) the sort of graph items that we shall use. Basically, we have sorts corresponding to those present in the  $\pi$ -calculus, i.e. processes ( $P$ ), prefixed processes ( $M$ ) and names ( $\mathcal{U}$ ) to which we add some auxiliary ones. More precisely, the node sorts we consider are  $\bullet$ ,  $\diamond$  and  $\circ$  that intuitively correspond to control points of processes and prefixed processes, and channel names. Edge labels are  $\tau$ ,  $\text{in}$  and  $\text{out}$  that respectively correspond to silent actions, inputs and outputs. Design labels are  $P$  and  $M$  with the obvious meaning. Our tentacles are also typed. We use a plain line and an arrow for the entry and exit control points, while channels and messages arguments of communication operations are denoted by concave- and bar-ended arrows.

We are now ready to define the graphical encoding of the  $\pi$ -calculus. We define it in terms of derived operators instead of using a functional encoding to stress the similarities and common sorting between the source calculus and the graph algebra.

**Definition 13 ( $\pi$  interpretation).** *The interpretation of the operators of the  $\pi$ -calculus over the design algebra is given by:*

$$\begin{array}{ll}
 \mathbf{0} \stackrel{\text{def}}{=} M_d[d] & N + O \stackrel{\text{def}}{=} M_d[N(d)|O(d)] \\
 (\nu x)R \stackrel{\text{def}}{=} P_p[(\nu x)R(p)] & \tau.Q \stackrel{\text{def}}{=} M_d[(\nu p)\tau(d,p)|Q(d)] \\
 Q \mid R \stackrel{\text{def}}{=} P_p[Q(p)|R(p)] & \bar{x}y.Q \stackrel{\text{def}}{=} M_d[(\nu p)\text{out}(d,x,y,p)|Q(d)] \\
 (P)Q \stackrel{\text{def}}{=} P_p[(\nu d)c(p,d)|Q(d)] & x(y).Q \stackrel{\text{def}}{=} M_d[(\nu p,y)\text{in}(d,x,y,p)|Q(d)]
 \end{array}$$



**Fig. 5.** Graphical encoding of process  $\bar{b}a.\bar{a}a \mid b(d).\bar{d}c$

together with axioms  $\text{flat}_P$  and  $\text{flat}_M$ .

The graphical representation of a representative subset of the above definition can be found in Figure 4. We remark only the most relevant aspects. Casting from sequential processes into parallel process must be done explicitly to choices and processes in parallel. This is done by connecting by preceding the graph of a sequential process with a  $c$ -labelled edge. The parallel composition of two processes  $X$  and  $Y$  amounts simply to embed the respective graphs of  $X$  and  $Y$  in  $P$ -typed edges attached to the same  $\bullet$ -typed node. The presence of the flattening axiom  $\text{flat}_P$  will dissolve the embedding and as a result the corresponding graphs will be at the same level. Processes in parallel mean thus graphs departing from the same control point. Another relevant part of the encoding regards the input prefix, since it involves a free name a bound one and a process. We see that the encoding of  $x(\bar{y}).Z$  consists of an arc representing the input operation which we attach to the main  $\diamond$ -typed control point and its  $\bullet$ -typed continuation where we plug the graph corresponding to  $Z$ . The edge representing the input action is connected to a free node representing the communication channel  $x$  and the restricted node representing the argument channel  $y$ .

It is worthwhile to remark that, although not formally stated here due to space limits, the proposed encoding is sound, i.e. equivalent processes are mapped into equivalent graphs. As a matter of fact the obtained graphs are the same as those proposed in the sound encoding of [9]. In addition our encoding characterises the set of all graphs that correspond to  $\pi$ -calculus processes, namely those generated by the derived algebra implicit in the encoding.

*Example 4.* As a concrete example consider process  $\bar{b}a.\bar{a}a \mid b(d).\bar{d}c$ , whose graphical encoding is depicted in Figure 5.

## 5 Graphs for 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 [1] for more detailed descriptions.

**Definition 14 (CaSPiS syntax).** Let  $\mathcal{R}$  be a set of session names,  $\mathcal{S}$  be a set of service names) and  $\mathcal{V}$  be a set of value names. A CaSPiS process  $P$  is a term generated by the syntax

$$\begin{aligned} P &::= M \mid r \triangleright P \mid r \triangleleft P \mid P > Q \mid (\nu w)P \mid P \mid P \\ M &::= \mathbf{0} \mid M + M \mid A.P \quad A ::= s \mid \bar{s} \mid (u) \mid (?x) \mid \langle u \rangle \mid \langle u \rangle^\dagger \end{aligned}$$

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  $\bar{s}.Q$ . Synchronisation of  $s.P$  and  $\bar{s}.Q$  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  $r \triangleright P$  and  $r \triangleleft Q$ , with  $r$  bound somewhere above them by  $(\nu r)$ . Values produced by  $P$  can be consumed by  $Q$ , and vice-versa: this permits description of interaction patterns more complex than the usual *one-way* and *request-response* typical of web services. Rules governing creation and scoping of sessions are based on those of the restriction operator in the  $\pi$ -calculus. Note that multiple invocations to the same persistent service will yield separate sessions and thus hierarchies of nested sessions. Values can be returned outside a session to the enclosing environment using the return operator,  $\langle \cdot \rangle^\dagger$ . 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  $P$ , not in a fresh one. It is important to remark that CaSPiS processes are considered up to the structural congruence  $\equiv_c$ .

**Definition 15 (CaSPiS congruence).** The structural congruence  $\equiv_c$  is the least congruence induced by  $\alpha$ -renaming (for restriction and abstraction) and the following laws:

$$\begin{array}{ll} P \mid (Q \mid R) \equiv (P \mid Q) \mid R & P \mid Q \equiv Q \mid P \\ P \mid \mathbf{0} \equiv P & ((\nu n)Q) > P \equiv (\nu n)(Q > P) \text{ if } n \notin \text{fn}(P) \\ (\nu n)\mathbf{0} \equiv \mathbf{0} & (\nu n)(P \mid Q) \equiv P \mid (\nu n)Q \text{ if } n \notin \text{fn}(P) \\ (\nu n)(\nu m)P \equiv (\nu m)(\nu n)P & r \triangleright (\nu n)P \equiv (\nu n)r \triangleright P \text{ if } n \neq r \\ (\nu n)A.P \equiv A.(\nu n)P & r \triangleleft (\nu n)P \equiv (\nu n)r \triangleleft P \text{ if } n \neq r \end{array}$$

*CaSPiS encoding.* We first define the alphabets of edge labels and nodes. The set  $\mathcal{NT}$  of design labels is composed by  $P$ ,  $M$ ,  $S$ ,  $D$ ,  $I$ ,  $F$  and  $T$  which respectively stand for Parallel processes, sequential processes (i.e. suMmations), Sessions, service Definitions, service Invocations and pipes (From and To). Sort  $T$  is further partitioned over  $2^{AP}$  (denoting each subsort as  $T^N$  to deal with a common problem when encoding replicated processes (the target process of a pipe is implicitly replicate for each value generated by the source). The set  $\mathcal{T}$  of edge labels contains  $c$  (explicit  $M$ -to- $P$  cast),  $d$  (service definition),  $k$  (service invocation),

in (abstraction),  $\langle \rangle$  (concretion) and **ret** (return). The node sorts considered are  $\circ$  (channels),  $\bullet$  (control points),  $*$  (service name, i.e.  $\mathcal{S}$ ) and  $\square$  (values, i.e.  $\mathcal{V}$ ). We assume that for each session name  $r$  there are two channel nodes  $r_{\triangleleft}, r_{\triangleright}$  representing the two directions of inter-session communication.

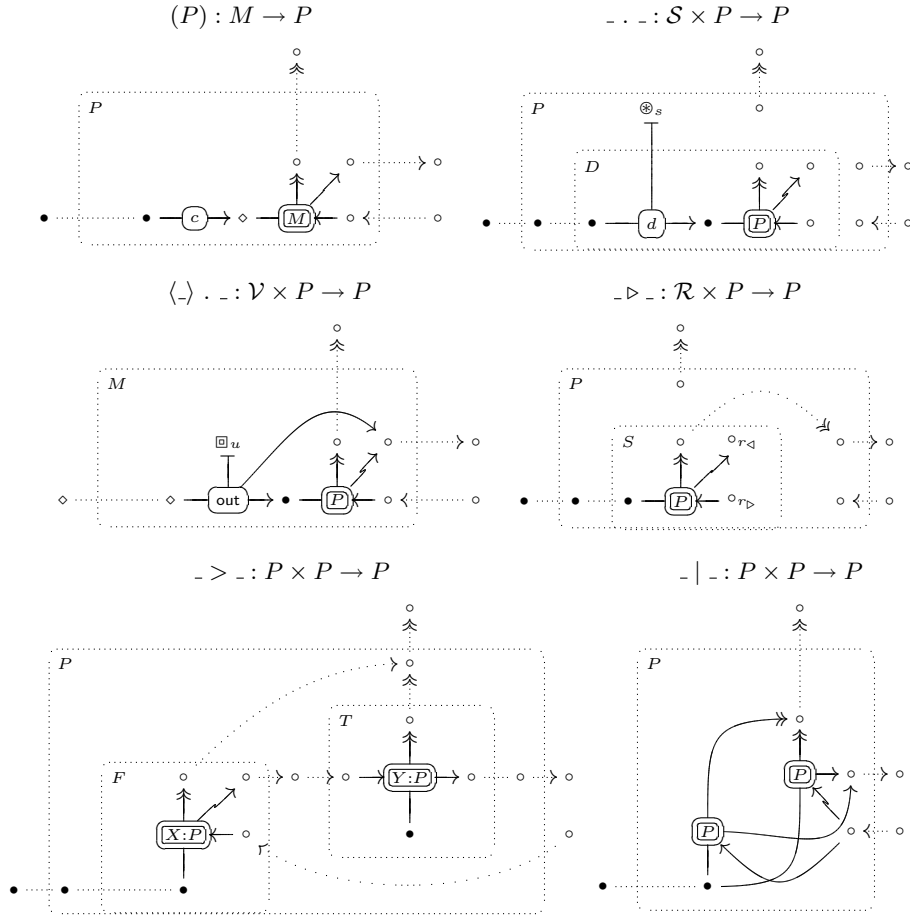
The graphical representation of each design and edge label and their respective types can be found in Figure 6. For instance, designs of type  $P$  are all of the form  $P_{(p,t,i,o)}[G]$  where  $p$  is the control point representing the process start of execution,  $t$  is the returning channel,  $i$  is the input channel and  $o$  is the output channel. Summations have the same kind of interface. Designs of type  $D$  and  $I$  only expose the starting point of execution, while pipes are very much like processes.

**Definition 16 (CaSPiS interpretation).** *The interpretation of CaSPiS constructors as derived operators of the design algebra is given by:*

$$\begin{aligned}
(P)M &\stackrel{\text{def}}{=} P_{(p,t,i,o)}[(\nu d)(c(p,d) \mid M(d,t,i,o))] \\
s.Q &\stackrel{\text{def}}{=} P_{(p,t,i,o)}[t \mid i \mid o \mid D_p[(\nu q,t,i,o)d(p,s,q) \mid Q(q,t,i,o)](p)] \\
\bar{s}.Q &\stackrel{\text{def}}{=} P_{(p,t,i,o)}[t \mid i \mid o \mid D_p[(\nu q,t,i,o)k(p,s,q) \mid Q(q,t,i,o)](p)] \\
r \triangleright Q &\stackrel{\text{def}}{=} P_{(p,t,i,o)}[i \mid o \mid S_{(p,t)}[Q(p,t,r_{\triangleright},r_{\triangleleft})](p,t)] \\
r \triangleleft Q &\stackrel{\text{def}}{=} P_{(p,t,i,o)}[i \mid o \mid S_{(p,t)}[Q(p,t,r_{\triangleleft},r_{\triangleright})](p,t)] \\
Q > R &\stackrel{\text{def}}{=} P_{(p,t,i,o)}[(\nu q,m)(F_{(p,t,i,o)}[Q(p,o,i,o)](p,t,i,m) \mid T_{(p,i,o)}^{fn(R)}[R(p,o,i,o)](q,m,o))] \\
Q \mid R &\stackrel{\text{def}}{=} P_{(p,t,i,o)}[Q(p,t,i,o) \mid R(p,t,i,o)] \\
(\nu u)Q &\stackrel{\text{def}}{=} P_{(p,t,i,o)}[(\nu u)Q(p,t,i,o)] \\
(\nu r)Q &\stackrel{\text{def}}{=} P_{(p,t,i,o)}[(\nu r)Q(p,t,i,o)] \\
\mathbf{0} &\stackrel{\text{def}}{=} P_{(p,t,i,o)}[p \mid t \mid i \mid o] \\
N + O &\stackrel{\text{def}}{=} M_{(d,t,i,o)}[N(d,t,i,o) \mid O(d,t,i,o)] \\
\langle u \rangle.P &\stackrel{\text{def}}{=} P_{(d,t,i,o)}[(\nu p)\text{out}(d,p,u,o) \mid Q(p,t,i,o)]
\end{aligned}$$

where we present the action of value concretion only, (further actions are defined similarly).

Part of the above definition is graphically represented in Figure 6. We explain our graphical notation of design operations in detail here. An edge is represented by a rounded box with its label inside. 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. Arguments of an operation are denoted by encircling the corresponding symbol. For instance, double boxes correspond to design variables, while node arguments of type  $\circ$ ,  $*$  and  $\square$  are represented by  $\odot$ ,  $\otimes$  and  $\boxplus$ , respectively. 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 unless preceded by their identity (as in the case of the service name argument in service definition and invocation). Our graphical notation tends to hide unnecessary details. For instance, argument names or the particular choice of



**Fig. 6.** Graphical representation of some CaSPiS interpreted operators.

hidden names are not always displayed. Argument names appear in the pipelining operation (bottom-left of Figure 6) to distinguish the two arguments of type  $P$ .

We consider all  $P$ - and  $M$ -typed designs to be flattened, i.e. we introduce flattening axioms  $\text{flat}_P$  and  $\text{flat}_M$  into  $\equiv_d$ , but not  $\text{flat}_S$ ,  $\text{flat}_D$ ,  $\text{flat}_I$ ,  $\text{flat}_E$ . As a consequence the only explicit hierarchies are introduced by session nesting, service definition, service invocation and pipelining. Flattening processes allows us to get rid of the associativity and commutativity of parallel composition and non-deterministic choices. The explicit embedding of sessions provides an intuitive visual representation.

We explain just a few representative operations in detail. The session operations are interpreted as a graph operation that wraps a process into 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 the session embedding hides the input and output channels of the embedded process: they are connected directly to the dedicated inter-communication nodes 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 to denote that it is a non-active process. Moreover, the 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 name 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  $(\nu w)(Q > R)$  and  $Q > (\nu w)R$  are not congruent CaSPiS processes, but neither are their corresponding graphs  $P_{(p,t,i,o)}[(\nu q, m, w)(F_{(p,t,i,o)}[Q(p, o, i, o)](p, t, i, m) \mid T_{(p,i,o)}^{fn(R)}[R(p, o, i, o)](q, m, o))]$  and  $P_{(p,t,i,o)}[(\nu m, w)(F_{(p,t,i,o)}[Q(p, o, i, o)](p, t, i, m) \mid T_{(p,i,o)}^{fn(R)}[(\nu w)R(p, o, i, o)](q, m, o))]$ . This happens because extruding  $w$  in the latter results in  $P_{(p,t,i,o)}[(\nu q, m, w)(F_{(p,t,i,o)}[Q(p, o, i, o)](p, t, i, m) \mid T_{(p,i,o)}^{fn(R) \cup w}[R(p, o, i, o)](q, m, o))]$  which is not equivalent to the former due to the different  $T$  subtypes.

*Example 5.* Let us now consider that the customer of the credit request scenario decides to consult two different financial institutions about their credit conditions and forward the first arrived response via an email service. In particular, one part of the scenario is specified by the following term:

$$\begin{aligned}
& (\nu \text{collector})(\text{collector}.\langle ?\text{credit} \rangle \langle \text{credit} \rangle^\dagger \\
& \quad \mid \overline{\text{collector}}.(\overline{\text{BankA}}.\langle ?\text{creditA} \rangle \langle \text{creditA} \rangle^\dagger \\
& \quad \mid (\overline{\text{BankB}}.\langle ?\text{creditB} \rangle \langle \text{creditB} \rangle^\dagger)) > (?msg).\overline{\text{emailMe}}.\langle msg \rangle
\end{aligned}$$

In words, a new service `collector` is defined and invoked to produce credit conditions pipelined into service `emailMe`. More precisely, service `collector` is invoked by a parallel process. Such processes, in turn, invoke services `BankA` and `BankB` to get credit conditions and pass the result to service `collector`, which will forward the credit condition by returning them with  $\langle \text{credit} \rangle^\dagger$ . Such return is captured by the pipeline and used as email message parameter to invoke service `emailMe`. The above term is supposed to be in parallel with the service definitions of the various credit services `BankA`, `BankB`. In the initial state no session is active. A first step can open a session between the collector definition and invocation, creating a session frame and the corresponding channels for inter-session communication. In a second step, the `BankB` credit service can be invoked creating a nested session, where the credit service is ready to communicate his current conditions. The resulting term (see Figure 7) is (where the bank's services part is neglected for simplicity)

$$\begin{aligned}
& (\nu \text{collector})((\nu r)r \triangleleft \langle ?\text{credit} \rangle \langle \text{credit} \rangle^\dagger \\
& \quad \mid r \triangleright (\overline{\text{BankA}}.\langle ?\text{creditA} \rangle \langle \text{creditA} \rangle^\dagger \\
& \quad \mid (\nu r).r \triangleleft \langle ?\text{creditB} \rangle \langle \text{creditB} \rangle^\dagger)) > (?msg).\overline{\text{emailMe}}.\langle msg \rangle
\end{aligned}$$

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

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

## 6 Related Work

Our technique is mainly inspired on the graphical encoding of the  $\pi$ -calculus presented in [9], an approach for the reconfiguration of software architectures called *Architectural Design Rewriting* (ADR) [3] and the graph algebra of [4].

The graphical representation of [9] is based on the DPO approach to graph transformation [5]. Each term of the  $\pi$ -calculus is represented by some special kind of normalized syntactic tree. The encoding is sound in the sense that two structurally equivalent processes are represented by equivalent (isomorphic) graphs. Moreover, each possible evolution of a process corresponds to the application (up to garbage collection) of a graph rewrite rule (and vice versa).

ADR [3] offers a formal setting where design development, run-time execution and reconfiguration are defined on the same foot by conciliating software architectures and process calculi via graphical methods. In particular, architectures are encoded as terms of a particular graph algebra and reconfigurations are defined using standard term rewriting techniques. Our graph algebra can be seen as a primitive algebra for ADR (i.e. original ADR specifications can be seen as particular derived algebras).

The algebra of [4] offers a compact and elegant way to represent graphs in an algebraic way. The main idea is to have constructors such as the empty graph, edges, and parallel composition, and axioms like associativity and commutativity of the parallel composition to consider graphs up to isomorphism. Our first graph algebra can be seen as an extension of the algebra of [4] to hierarchical graphs and as an extension of ADR with names, which enables a more suitable representation of nominal calculi and thus ordinary behaviour.

Our approach is closely related to other approaches that propose graphical representation of concurrent systems such as those based on the DPO approach to graph transformation [9], Bigraphs [11] or Synchronized Hyperedge Replacement [8]. Over such approaches ours offers a simpler syntax that facilitates graphical encodings.

We do not promote our work as an alternative to such valid approaches. Instead, we believe that our approach can be useful as an intermediate language between process calculi and their concrete graphical models. In this work we have chosen a particular domain of hierarchical graphs but it is in principle possible to equip our graph algebra with an interpretation over other models such as the above mentioned ones.

## 7 Conclusion

We presented a novel specification formalism based on hierarchical graphs particularly suited to define graphical encodings of service oriented calculi which

typically exhibit hierarchical features such as sessions and transactions. Our approach was illustrated by presenting a graphical encoding of the  $\pi$ -calculus roughly equivalent to an existing one [9] and a novel graphical encoding of a recently proposed session-centered calculus.

We believe that our approach can serve as a useful inspiration to equip well-known graphical models of communication with syntax notations that facilitate the definition of intuitive and correct encodings of process calculi. We remark that we have restricted our examples to the finite fragments of the corresponding calculi. Dealing with infinite fragments in form of recursive definitions or replication is involved. Some solutions exist, based for instance in rewrite rules for the expansion of process definitions [9]. This is subject of current work.

We plan to apply our technique to other process calculi, with a particular focus on other service oriented calculi. Implementing our approach in the prototypical implementation of ADR [2] or some other graph rewrite engine is also planned.

## References

1. M. Boreale, R. Bruni, R. D. Nicola, and M. Loreti. Sessions and pipelines for structured service programming. In *Proceedings of the 10th IFIP International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS'08)*, volume 5051 of *LNCS*. Springer, 2008.
2. R. Bruni, A. Lluch Lafuente, and U. Montanari. Hierarchical Design Rewriting with Maude. In *Proceedings of the 7th International Workshop on Rewriting Logic and its Applications (WRLA'08)*, ENTCS. Elsevier, 2008. To appear.
3. R. Bruni, A. Lluch Lafuente, U. Montanari, and E. Tuosto. Style Based Architectural Reconfigurations. *Bulletin of the European Association for Theoretical Computer Science (EATCS)*, (94):161–180, February 2008.
4. A. Corradini, U. Montanari, and F. Rossi. An abstract machine for concurrent modular systems: CHARM. *Theoretical Computer Science*, 122(1&2):165–200, 1994.
5. A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. Algebraic approaches to graph transformation - part i: Basic concepts and double pushout approach. In *Handbook of Graph Grammars*, pages 163–246, 1997.
6. F. Drewes, B. Hoffmann, and D. Plump. Hierarchical graph transformation. *J. Comput. Syst. Sci.*, 64(2):249–283, 2002.
7. F. Drewes, H.-J. Kreowski, and A. Habel. Hyperedge replacement, graph grammars. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*, pages 95–162. World Scientific, 1997.
8. G. L. Ferrari, D. Hirsch, I. Lanese, U. Montanari, and E. Tuosto. Synchronised hyperedge replacement as a model for service oriented computing. In *4th International Symposium on Formal Methods for Components and Objects (FMCO'05)*, volume 4111 of *LNCS*, pages 22–43. Springer, 2006.
9. F. Gadducci. Graph rewriting for the  $\pi$ -calculus. *Mathematical Structures in Computer Science*, 17(3):407–437, 2007.
10. D. Grohmann and M. Miculan. An algebra for directed bigraphs. *Electr. Notes Theor. Comput. Sci.*, 203(1):49–63, 2008.
11. O. H. Jensen and R. Milner. Bigraphs and mobile processes. Technical Report 570, Computer Laboratory, University of Cambridge, 2003.

12. R. Milner. *Communicating and Mobile Systems: The  $\pi$ -calculus*. Cambridge University Press, 1992.

## A Proofs

**Theorem 1. [design interpretation]** *Let  $t_1, t_2$  be terms (graph or designs) generated by the design algebra. Then,  $t_1 \equiv_d t_2$  if and only if  $\llbracket t_1 \rrbracket \equiv_g \llbracket t_2 \rrbracket$ .*

*Proof.* Soundness of the interpretation of designs is done by showing that for each axiom of  $\equiv_d$  (see Definition 4) we have that the left- and right-hand sides are interpreted as equivalent graphs (according to  $\equiv_g$ ). The proof of completeness is done by structural induction on the normal form of designs. The normal form of designs is basically an extension of the normal form for the terms of the graph algebra of [4]. Roughly, all restricted nodes are shifted. The normal form can be obtained as in [4] adding a left-to-right reading to the axioms of removal of present nodes and node-extrusion over nesting.

**Theorem 2. [caspis interpretation]** *For any two processes  $P$  and  $Q$  we have  $P \equiv_c Q$  iff  $P \equiv_d Q$ .*

*Proof. (Sketch)* Soundness of our encoding is reduced to show that for each axiom of CaSPiS congruence  $\equiv_c$  (see Definition 15) we have that the left- and right-hand sides are interpreted as equivalent designs terms (according to  $\equiv_d$ ). The proof for AC1 axioms for parallel and non-deterministic composition of processes in  $\equiv$  is straightforward as we have similar axioms for parallel composition of graphs in  $\equiv_d$ . Idem can be said for the axioms regarding the order of name restrictions and restriction for an empty process as we have equivalent axioms for node restriction in our design algebra. Let us now consider name extrusion for pipelines, services and actions. For axiom  $((\nu n)Q) > P \equiv (\nu n)(Q > P)$  we observe that both sides are interpreted as  $P_{(p,t,i,o)}[(\nu u)P(p,t,i,o) \mid Q(p,t,i,o)]$  (after flattening). The proof for name extrusion in sessions is similar but based on flattening and node extrusion for designs. Moving name restriction over action prefixes is also similarly shown.

The proof of completeness is done by structural induction on the normal form of processes. The normal form of CaSPiS processes is  $(\nu W)(\prod_i R_i > Q_i \mid \prod r_i \bowtie S_i \mid \prod_i \sum_j A_{i,j}.M_{i,j})$ . Assume that the encoding is not complete, i.e. that we have two processes  $P, P'$  that are not equivalent but are interpreted as equivalent designs, i.e. we have  $P \not\equiv_c P'$  but  $P \equiv_d P'$ . Let us denote their normal forms by  $(\nu W)(\prod_i R_i > Q_i \mid \prod r_i \bowtie S_i \mid \prod_i \sum_j A_{i,j}.M_{i,j})$  and  $(\nu W')(\prod_i R'_i > Q'_i \mid \prod R'_i \bowtie S'_i \mid \prod_i \sum_j A'_{i,j}.M'_{i,j})$ , respectively.

## B The $\pi$ -calculus

This section recalls the basics the  $\pi$ -calculus. We restrict our work to the finite fragment of the calculus. Some differences with the standard presentation is that we separate prefixed processes from parallel processes and we present actions and prefixing as unique combined ternary operators. The distinction enables a more elegant presentation of our approach.

**Definition 17 (processes).** Let  $\mathcal{U}$  be a set of names. A process  $P$  is a term generated by the syntax

$$\begin{aligned} P &::= M \mid (\nu a)P \mid P \mid P \\ M &::= \mathbf{0} \mid M + M \mid A.P \\ A &::= \tau.P \mid a(b).P \mid a(\bar{b}).P \end{aligned}$$

where  $a, b \in \mathcal{U}$ .

In the definition above, terms generated by  $P$  and  $M$  are respectively called *process*, *prefixed processes* (or *summations*). The standard definition for the set of free names of a process  $P$ , denoted by  $fn(P)$ , is assumed. Similarly for  $\alpha$ -convertibility, with respect to the *restriction* operators  $(\nu a)P$ , the *input* operators  $a(b).P$ : in all cases, the name  $a$  is bound in  $P$ , and it can be freely  $\alpha$ -converted.

Using the definition above, the behaviour of a process  $P$  is described as a relation over *abstract processes*, i.e. a relation closed under structural congruence.

**Definition 18 (structural congruence).** The structural congruence for processes is the relation  $\equiv_{\subseteq} \mathcal{P} \times \mathcal{P}$ , closed under process construction and  $\alpha$ -conversion, inductively generated by the following set of axioms

$$\begin{aligned} P \mid Q &\equiv Q \mid P & P \mid (Q \mid R) &\equiv (P \mid Q) \mid R & P \mid \mathbf{0} &\equiv P \\ M + N &\equiv N + M & M + (N + O) &\equiv (M + N) + O & M + \mathbf{0} &\equiv M \\ (\nu a)\mathbf{0} &\equiv \mathbf{0} & (\nu a)(\nu b)P &\equiv (\nu b)(\nu a)P & (\nu a)(P \mid Q) &\equiv P \mid (\nu a)Q \text{ if } a \notin fn(P) \\ (\nu a)A.P &\equiv A(\nu a).P \text{ if } a \notin fn(A), A \in \{\tau, a(b), a(\bar{b})\} \end{aligned}$$

As usual,  $P\{Q/x\}$  denotes process  $P$  after the capture-avoiding substitution of each free occurrence of process variable  $x$  with process  $Q$ .

### C Graphs for CaSPiS

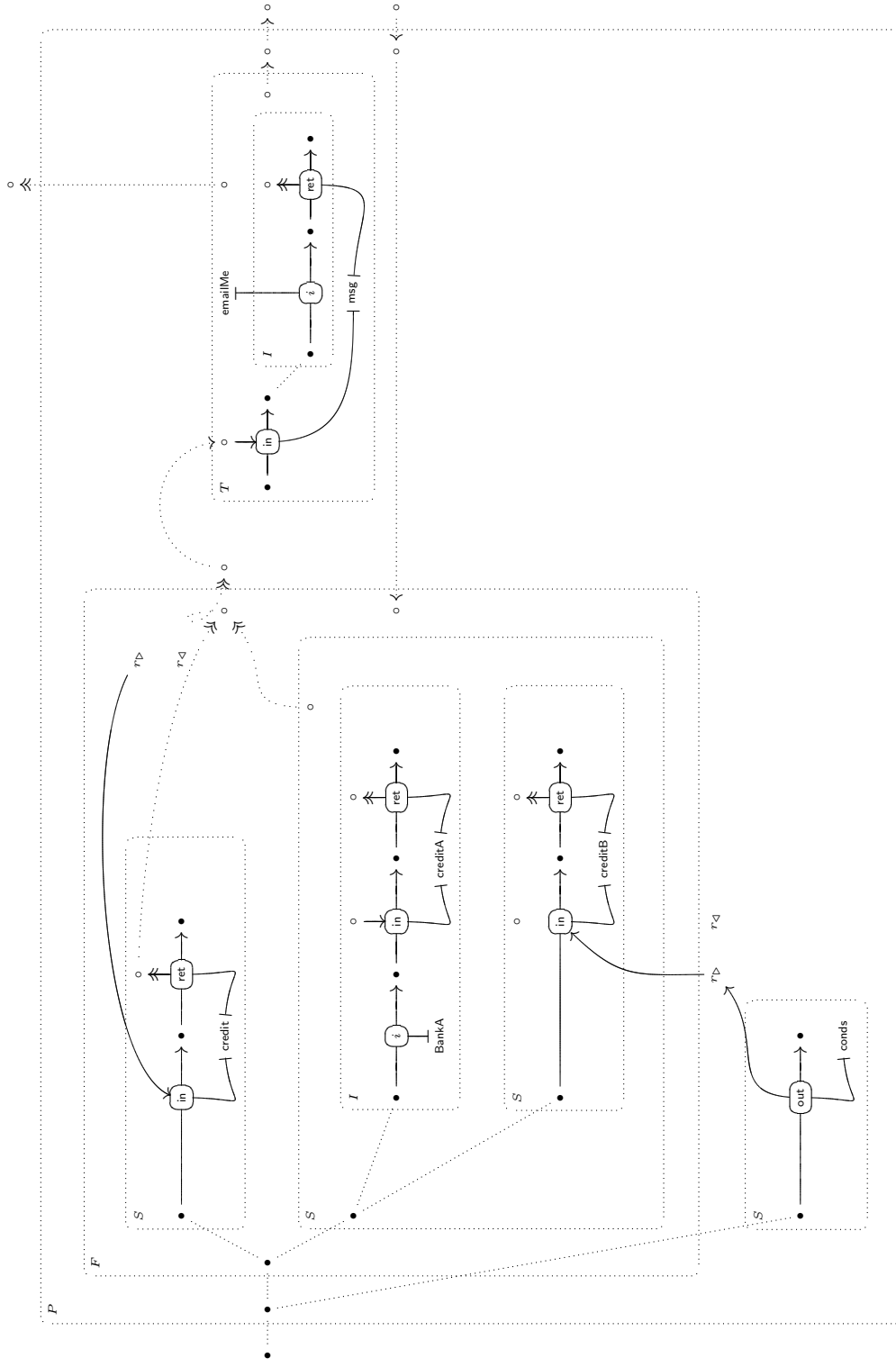


Fig. 7. Graphical representation of credit collector service after session creation.