Hierarchical Models for Service-oriented Systems*

Roberto Bruni[1], Andrea Corradini[1],
Fabio Gadducci[1], Alberto Lluch Lafuente[2], and Ugo Montanari[1]

[1] Department of Computer Science, University of Pisa, Italy
{bruni,andrea,gadducci,ugo}@di.unipi.it
[2] IMT Institute for Advanced Studies Lucca, Italy
alberto.lluch@imtlucca.it

Abstract. We present our approach to the denotation and representation of hierarchical graphs: a suitable algebra of hierarchical graphs and two domains of interpretations. Each domain of interpretation focuses on a particular perspective of the graph hierarchy: the top view (nested boxes) is based on a notion of embedded graphs while the side view (tree hierarchy) is based on gs-graphs. Our algebra can be understood as a high-level language for describing such graphical models, which are well suited for defining graphical representations of service-oriented systems where nesting (e.g. sessions, transactions, locations) and linking (e.g. shared channels, resources, names) are key aspects.

1 Introduction

As witnessed by a vast literature, graphs offer a convenient ground for the specification and analysis of software systems. As an example, the use of graphs as a suitable domain for the visualisation of a system specified by algebraic means is pursued in various proposals, based on traditional Graph Transformation [15], Bigraphical Reactive Systems [16], and Synchronised Hyperedge Replacement [13].

Despite their expressiveness and flexibility, the use of these formalisms to build a graphical representation for an existing specification language involves two major challenges. First, encoding system configurations (states), guaranteeing that structural equivalence is preserved: i.e. equivalent (e.g. structurally congruent) configurations are mapped into equivalent (e.g. isomorphic) graphs. Second, encoding system dynamics (e.g. behaviour, reconfigurations, model transformations, refactorings), guaranteeing that the original semantics is respected.

Preserving structural equivalence has several advantages. It offers an intuitive normal form representation for systems, and it allows us to reuse results and techniques from graph theory for solving specific problems. In particular, the soundness of the encoding is necessary to use graph transformation approaches [10] to model dynamic aspects since (sub)graph isomorphism is at the base of the rule matching mechanism.

---

* This work has been partially sponsored by the project SENSORIA, IST-2005-016004.

The encoding of configurations given with an algebraic syntax (e.g. as in process calculi) is facilitated by their structure (i.e. processes are terms) since it can be defined inductively. In absence of an algebraic presentation for the language under consideration, ad-hoc algebraic syntax must be developed if one wants to benefit from structural induction in proofs, transformations or definitions. Still, most graph models are not equipped with algebraic syntax and those that exist require advanced skills to deal with sophisticated models involving set-theoretic definitions of graphs with interfaces (e.g. [15]) or complex type systems (e.g. [7]), hampering definitions and proofs. Moreover, one encounters a severe drawback: namely, the syntax of graph formalisms are often very different from the source language and not provided with suitable primitives to deal with features that commonly arise in algebraic specifications, like names (e.g. references, channels), name restrictions (e.g. hiding, nonce generation) or hierarchical aspects (e.g. ambients, scopes) in the case of process calculi. Identifying the right structure is fundamental to provide scalable techniques.

Our goal is to define a simple flexible syntax for hierarchical models and to develop a technique that simplifies the definition of graphical representations of languages. We think that nesting and linking must be treated as first-class concepts, conveniently represented with a suitable syntax that allows one to express and exploit them. Nesting and linking are two key structural aspects that arise repeatedly in computer systems: consider e.g. the structure of file systems, composite diagrams, networks, membranes, sessions, transactions, locations, structured state machines or XML files. In particular, nesting plays a fundamental role for abstracting the complexity of a system by offering different levels of detail. Various graphical models of nesting and sharing structures already exist but (as we claim in [3, 5, 4]) none of them offer a simple, intuitive syntax.

Here, the gap between the different levels of abstraction at which algebraic specifications and graphical models reside is filled by a simple algebra that enjoys primitives for dealing with names, restriction, parallel composition and, most importantly, nesting and that is equipped with a (sound and complete) set of axioms equating two terms whenever they represent isomorphic graphs. Besides facilitating the visual specification of configurations, the algebraic structure facilitates definitions, transformations and proofs by induction.

Structure of this chapter. § 2 introduces the algebra of hierarchical graphs. § 3 presents our two models of hierarchical graphs. § 4 shows the expressiveness and flexibility of our design algebra in modelling heterogeneous notations, ranging from workflow languages to sophisticated process calculi.

2 The Syntax of Hierarchical Graphs

We introduce our algebra of hierarchical graphs that we call *designs*. The algebraic presentation of designs is mostly inspired by the graph algebra of [9].

As a matter of notation, we let $\lfloor \overline{x} \rfloor$ denote the set of elements of a list $\overline{x}$ and, conversely, $\lceil X \rceil$ the vector of elements of an ordered set X. We overload $|\cdot|$ to denote both the length of a list and the cardinality of a set.

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

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

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

Terms generated by $\mathbb{G}$ and $\mathbb{D}$ are meant to represent (possibly hierarchical) graphs and "edge-encapsulated" hierarchical graphs, respectively. The syntax has the following informal meaning: $\mathbf{0}$ represents the empty graph, x is a discrete graph containing node x only, $l\langle\overline{x}\rangle$ is a graph formed by an l-labelled (hyper)edge attached to nodes $\overline{x}$ (the i-th tentacle to the i-th node in $\overline{x}$, sometimes denoted by $\overline{x}[i]$), $\mathbb{G} \mid \mathbb{H}$ is the graph resulting from the parallel composition of graphs $\mathbb{G}$ and $\mathbb{H}$ (their disjoint union up to shared nodes), $(\nu x)\mathbb{G}$ is the graph $\mathbb{G}$ after making node x not visible from the outside (borrowing nominal calculus jargon we say that the node x is *restricted*), and $\mathbb{D}\langle\overline{x}\rangle$ 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 $L_{\overline{x}}[\mathbb{G}]$ is a design labelled by L, with body graph $\mathbb{G}$ whose nodes $\overline{x}$ are exposed in the interface. To clarify the exact role of the interface of a design, we can use a programming metaphor: a design $L_{\overline{x}}[\mathbb{G}]$ is like a procedure declaration where $\overline{x}$ is the list of formal parameters. Then, term $L_{\overline{x}}[\mathbb{G}]\langle\overline{y}\rangle$ represents the application of the procedure to the list of actual parameters $\overline{y}$; of course, in this case the lengths of $\overline{x}$ and $\overline{y}$ must be equal (more precisely, the applicability of a design to a list of nodes must satisfy other requirements to be detailed later in the definition of well-formedness). In the following, we shall often write $L[\mathbb{G}]\langle\overline{y}\rangle$ as a shorthand for $L_{\overline{y}}[\mathbb{G}]\langle\overline{y}\rangle$.

Restriction $(\nu x)\mathbb{G}$ acts as a binder for x in $\mathbb{G}$ and similarly $L_{\overline{x}}[\mathbb{G}]$ binds $\overline{x}$ in $\mathbb{G}$. As usual, restrictions and interfaces lead to the notion of *free* nodes.

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

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

Example 1. Let $a, b \in \mathcal{E}$, $A \in \mathcal{D}$, $u, v, w, x, y \in \mathcal{N}$. We write and depict in Fig. 1 some terms of our algebra, where for helping intuition an informal, appealing visual notation is preferred to the formal underlying graphs that will be described in [5]. Nodes are represented by circles, edges by small rounded boxes, and designs by large shaded boxes with a top bar. 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. In the particular examples only free nodes are annotated with their identities, while restricted nodes are anonymous (no label). Note how the tentacles of a- and b-labelled boxes attached to x and y do actually cross the interface and are hence denoted by small black boxes in border of the A-labelled designs. This does not happen for tentacles attached to w since it is shared node.

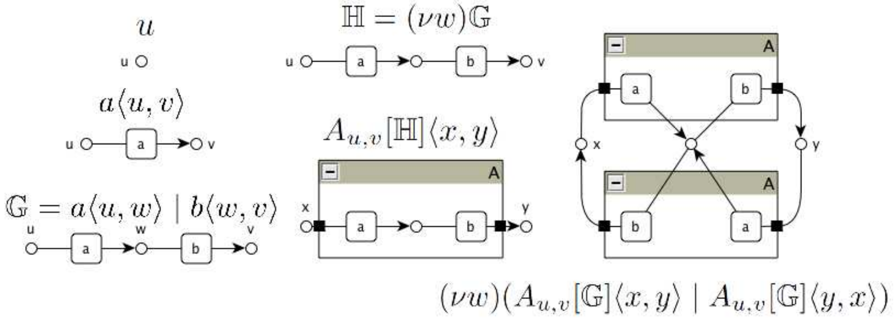
u
u
$a\langle u, v\rangle$
u a v
$\mathbb{H} = (\nu w)\mathbb{G}$
u a b v
$A_{u,v}[\mathbb{H}]\langle x, y\rangle$
A
x a b y
$\mathbb{G} = a\langle u, w\rangle \mid b\langle w, v\rangle$
u w v
a b
A a b
x y
A b a
$(\nu w)(A_{u,v}[\mathbb{G}]\langle x, y\rangle \mid A_{u,v}[\mathbb{G}]\langle y, x\rangle)$

Fig. 1. Some terms of the graph algebra and their informal visual notation

In practice, it is very frequent that one is interested in disciplining the use of edge and design labels so to be attached only to a specific number of nodes (possibly of specific sorts) or to contain graphs of a specific shape. To this aim it is typically the case that: 1) nodes are sorted, in which case their labels take the form $x : s$ for $x \in \mathcal{X}$ the *name* and $s \in S$ the *sort* of the node; 2) each label $l \in \mathcal{E}$ (resp. $L \in \mathcal{D}$) has a fixed rank denoted $ar(l) \in S^*$ (resp. $ar(L) \in S^*$); 3) designs can be partitioned according to their top-level labels (i.e. the set of design labels $\mathcal{D}$ can be seen as the set of sorts, with a membership predicate $\mathbb{D} : L$ that holds whenever $\mathbb{D} = L_{\overline{x}}[\mathbb{G}]$ for some $\overline{x}$ and $\mathbb{G}$).

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 that the (vectors of) types of $\overline{x}$ and L coincide, and similarly for typed operators $\mathbb{D}\langle\overline{x}\rangle$ and $l(\overline{x})$. From now on, we restrict our attention to well-formed designs: all the axioms are going to preserve well-formedness and all the derived operators used for the encodings are well-formed.

Definition 3 (well-formedness). *A well-typed design or graph is* well-formed *if:*

1. *for each occurrence of design $L_{\overline{x}}[\mathbb{G}]$ we have $\lfloor\overline{x}\rfloor \subseteq fn(\mathbb{G})$;*
2. *for each occurrence of graph $L_{\overline{x}}[\mathbb{G}]\langle\overline{y}\rangle$, the substitution ${}^{\overline{x}}/_{\overline{y}}$ is a function.*

Intuitively, the restriction on the mapping ${}^{\overline{x}}/_{\overline{y}}$ allows $\overline{x}$ to account for matching of nodes in the interface: distinct nodes in $\overline{y}$ must correspond to distinct nodes in $\overline{x}$ (as the list $\overline{x}$ can contain repetitions).

In order to have a notion of "structurally equivalent" designs, the algebra includes the structural graph axioms of [9] such as associativity and commutativity for $\mid$ with identity $\mathbf{0}$ (axioms DA1–DA3 in Definition 4) and name extrusion (DA4–DA6). In addition, it includes axioms to α-rename bound nodes (DA7–DA8), an axiom for making immaterial the addition of a node to a graph where that same node is already free (DA9) and another one ensuring that global names are not localised within hierarchical edges (DA10).

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

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

where in axiom (DA7) *the substitution is required to be a function (to avoid node coalescing) and to respect the typing (to preserve well-formedness).*

It is immediate to observe that structural congruence respects free nodes, i.e. $\mathbb{G} \equiv_{\mathrm{D}} \mathbb{H}$ implies $\textit{fn}(\mathbb{G}) = \textit{fn}(\mathbb{H})$ for any $\mathbb{G}, \mathbb{H}$. Moreover, being $\equiv_{\mathrm{D}}$ a congruence, we remark, e.g. that $L_{\overline{x}}[\mathbb{G}] \equiv_{\mathrm{D}} L_{\overline{x}}[\mathbb{H}]$ whenever $\mathbb{G} \equiv_{\mathrm{D}} \mathbb{H}$.

One important aspect of our algebra is allowing the derivation of standard representatives for the equivalence classes induced by $\equiv_{\mathrm{D}}$.

Definition 5 (Normalized form). *A term* $\mathbb{G}$ *is in* normalised form *if it is* $\mathbf{0}$ *or it has the shape (for some* $n + m + p + q \geq 1$ *and suitable nodes* x_j*,* z_k *and edges* $l_h\langle \overline{v}_h \rangle$*,* $L^i_{\overline{y}_i}[\mathbb{G}_i]\langle \overline{w}_i \rangle$*):*

$$(\nu x_1) \ldots (\nu x_m)(\, z_1 \mid \ldots \mid z_n \mid l_1\langle \overline{v}_1 \rangle \mid \ldots \mid l_p\langle \overline{v}_p \rangle \mid L^1_{\overline{y}_1}[\mathbb{G}_1]\langle \overline{w}_1 \rangle \mid \ldots \mid L^q_{\overline{y}_q}[\mathbb{G}_q]\langle \overline{w}_q \rangle\,)$$

where all terms $\mathbb{G}_i$ *are in normalised form, all nodes* x_j *are pairwise distinct, all nodes* z_k *are pairwise distinct and letting* $X = \{x_1, \ldots, x_m\}$ *and* $Z = \{z_1, \ldots, z_n\}$ *we have* $X \subseteq Z$*,* $\textit{fn}(\mathbb{G}) = Z \setminus X$ *and* $\textit{fn}(L^i_{\overline{y}_i}[\mathbb{G}_i]\langle \overline{w}_i \rangle) = Z$ *for all* $i = 1...q$ *.*

Proposition 1. *Any term* $\mathbb{G}$ *admits a* $\equiv_{\mathrm{D}}$*-equivalent term* $\textit{norm}(\mathbb{G})$ *in normalised form.*

Roughly, in $\textit{norm}(\mathbb{G})$ the top-level restrictions are grouped to the left, and all the global names z_k are made explicit and propagated inside each single component $L^i_{\overline{y}_i}[\mathbb{G}_i]\langle \overline{w}_i \rangle$. Up to α-renaming and to nodes and edges permutation, the normalised form is actually proved to be unique.

3 The Models of Hierarchical Graphs

In this section we present our two models of hierarchical graphs.

3.1 Top-view Model

In [5] we have defined a new, suitable notion of hierarchical graphs with interface: roughly they extend ordinary hyper-graphs with the possibility to embed (recursively) a hierarchical graph within each edge, thus inducing a layered structure of nodes and edges. Notably, the nodes defined in one layer are also visible below in the hierarchy (but not above). The main result of [5] is to show that the encoding of design terms in hierarchical graphs is surjective and that the axiomatisation of the design algebra is sound and complete w.r.t. the encoding. Moreover, in the presence of flattening- or extrusion-axioms (see § 4.1) the encoding can be slightly modified so to extend the validity of main results. The drawing of hierarchical graphs as defined in [5] is along the informal drawing seen in Fig. 1: to some extent they illustrate a top view of the system.

We first present the set of *plain* graphs and graph *layers*, upon which we build our novel notion of *hierarchical* graphs. In the following, $\mathcal{N}$ and $\mathcal{A} = \mathcal{A}_{\mathcal{E}} \uplus \mathcal{A}_{\mathcal{D}}$ denote the universe of nodes and edges, respectively, for $\mathcal{A}$ indexed over the vocabularies $\mathcal{E}$ and $\mathcal{D}$.

Definition 6 (graph layer). *The set $\mathcal{L}$ of* graph layers *is the set of tuples $G = \langle N_G, E_G, t_G, F_G \rangle$ where $E_G \subseteq \mathcal{A}$ is a (finite) set of edges, $N_G \subseteq \mathcal{N}$ a (finite) set of nodes, $t_G : E_G \to N_G^*$ a tentacle function, and $F_G \subseteq N_G$ a set of free nodes. The set $\mathcal{P}$ of* plain graphs *contains those graph layers G such that $E_G \subseteq \mathcal{A}_{\mathcal{E}}$.*

Thus, we just equipped the standard notion of hypergraph with a chosen set of *free* nodes, intuitively denoting those nodes that are available to the environment, mimicking free names of our algebra. Next, we build the set of hierarchical graphs.

Definition 7 (hierarchical graph). *The set $\mathcal{H}$ of* hierarchical graphs *is the smallest set*[3] *containing all the tuples $G = \langle N_G, E_G, t_G, i_G, x_G, r_G, F_G \rangle$ where*

1. *$\langle E_G, N_G, t_G, F_G \rangle$ is a graph layer,*
2. *$i_G : E_G \cap \mathcal{A}_{\mathcal{D}} \to \mathcal{H}$ is an embedding function (we say $i_G(e)$ is the* inner *graph of $e \in E_G \cap \mathcal{A}_{\mathcal{D}}$),*
3. *$x_G : E_G \cap \mathcal{A}_{\mathcal{D}} \to \mathcal{N}^*$ is an exposure function ($x_G(e)$ tells which nodes of $i_G(e)$ are exposed and in which order), such that for all $e \in E_G \cap \mathcal{A}_{\mathcal{D}}$*
    - *(a) $\lfloor x_G(e) \rfloor \subseteq N_{i_G(e)} \setminus F_{i_G(e)}$, i.e. free nodes of inner graphs are not exposed;*
    - *(b) $|x_G(e)| = |t_G(e)|$, i.e. exposure and tentacle functions have the same arity;*[4]
    - *(c) $\forall n, m \in \mathbb{N}$ if $x_G(e)[n] = x_G(e)[m]$ then $t_G(e)[n] = t_G(e)[m]$, i.e. it is not possible to expose a node twice without attaching it to the same external node.*
4. *$r_G : E_G \cap \mathcal{A}_{\mathcal{D}} \to (N_G \hookrightarrow \mathcal{N})$ is a renaming function ($r_G(e)$ tells how nodes N_G are named in $i_G(e)$), such that for all $e \in E_G \cap \mathcal{A}_{\mathcal{D}}$ $r_G(e)(N_G) = F_{i_G(e)}$, i.e. the nodes of the graph are (after renaming) the free nodes of inner layers.*

Thus, a hierarchical graph G is either a plain graph, or it is equipped with a function associating to each edge in $E_G \cap \mathcal{A}_{\mathcal{D}}$ another graph. The tuple $\langle N_G, E_G, t_G, i_G \rangle$ recalls the layered model of hierarchical graphs of [11], with i_G

---

[3] Taking the least set we exclude cyclic dependencies from containment, like a graph being embedded in one of its edges.

[4] We shall not put any emphasis on the typing of the graph, but clearly if the set of nodes is many sorted an additional requirement should force the exposure and tentacle functions to agree on the node types.

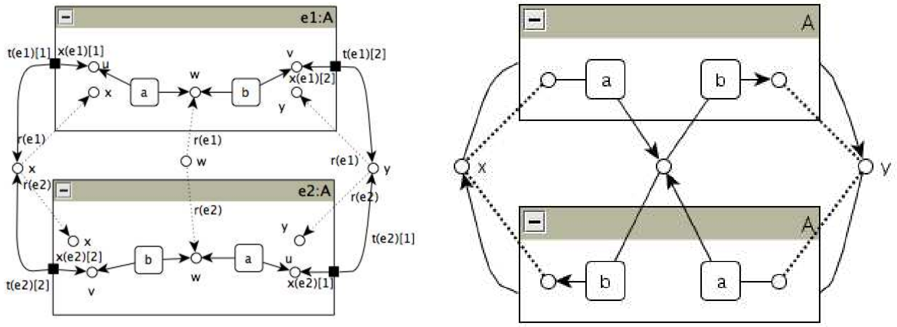

Fig. 2. A hierarchical graph (left) and its simplified representation (right)

being the function that embeds a graph (of a lower layer) inside an edge. Node sharing is introduced by the graph component F_G and the renaming function r_G, inspired by the graphs with (cospan-based) interfaces of [15]. In practice, we shall often assume that $r_G(e)$ (when defined) is the ordinary inclusion: the general case is useful for embedding (and reuse) graphs without renaming their nodes.

Example 2. Consider the last term of Example 1 and its informal graphical representation on Fig. 1 (right). Its actual interpretation as a hierarchical graph appears in Fig. 2 (left) decorated with the most relevant annotations (the tentacle, exposition and renaming functions for the two hierarchical edges). As witnessed by Fig. 2 (right), we can introduce convenient shorthands, such as dotted lines for mapping parameters, node-sharing represented by unique nodes and tentacles crossing the hierarchy levels, dropping the order of tentacles in favour of graphical decorations (missing or different heads and tails) to get a simplified notation that still retains all the relevant information. Note that such a simplified representation is very close to the informal notation shown in Fig. 1.

The above example should highlight that the algebra is providing a simple syntax that hides the complexities of our hierarchical model. The syntax can then be used in definitions, proofs and transformations in a much more friendly way than would be the case when working directly with the actual graphs.

We now present the interpretation of terms as graphs. In the definition below we assume that subscripts refer to the corresponding encoded graph. For instance, $[\![\mathbb{G}]\!] = \langle N_{\mathbb{G}}, E_{\mathbb{G}}, t_{\mathbb{G}}, i_{\mathbb{G}}, x_{\mathbb{G}}, r_{\mathbb{G}}, F_{\mathbb{G}} \rangle$.

Definition 8 (graph interpretation). *The encoding $[\![\cdot]\!]$, mapping well-formed terms into graphs, is the function inductively defined as*

$$\begin{array}{rlrl}
[\![x]\!] &= \langle \{x\}, \emptyset, \bot, \bot, \bot, \bot, \{x\} \rangle & [\![l\langle \overline{x} \rangle]\!] &= \langle \lfloor \overline{x} \rfloor, \{e'\}, e' \mapsto \overline{x}, \bot, \bot, \bot, \lfloor \overline{x} \rfloor \rangle \\
[\![\mathbb{G} \mid \mathbb{H}]\!] &= [\![\mathbb{G}]\!] \oplus [\![\mathbb{H}]\!] & [\![\mathbf{0}]\!] &= \langle \emptyset, \emptyset, \bot, \bot, \bot, \bot, \emptyset \rangle \\
[\![(\nu x)\mathbb{G}]\!] &= \langle N_{\mathbb{G}}, E_{\mathbb{G}}, t_{\mathbb{G}}, i_{\mathbb{G}}, x_{\mathbb{G}}, r_{\mathbb{G}}, F_{\mathbb{G}} \setminus \{x\} \rangle & & \\
[\![L_{\overline{x}}[\mathbb{G}]\langle \overline{y} \rangle]\!] &= \multicolumn{3}{l}{\langle N_{\mathbb{G}}, \{e\}, e \mapsto \overline{y}, e \mapsto [\![\mathbb{G}]\!] \oplus [\![\lfloor \overline{y} \rfloor]\!], e \mapsto \overline{x}, e \mapsto id_N, (F_{\mathbb{G}} \setminus \lfloor \overline{x} \rfloor) \cup \lfloor \overline{y} \rfloor \rangle}
\end{array}$$

where $e' \in \mathcal{A}_{\mathcal{E}}$ and $e \in \mathcal{A}_{\mathcal{D}}$, $\bot$ denotes the empty function, and $G \oplus H$ is a graph composition operation that build the disjoint union of G and H up to their common free nodes (see [5] for the full definition).

The encoding into (plain) graphs of the empty design, isolated nodes and single edges is trivial. Node restriction consists of removing the restricted node from the set of free nodes. The encoding of the parallel composition is as expected: a disjoint union of the corresponding hierarchical graphs up to common free nodes, plus a possible saturation of the sub-graphs with the nodes now appearing in the top graph layer. A hierarchical edge (last two rows) is basically a graph with 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), while adding the names $\lfloor \overline{y} \rfloor$ among the free ones.

The main result in [5] shows that the encoding is sound and complete, meaning that equivalent terms are mapped to isomorphic graph (and vice versa).

Theorem 1 (cf. [5]). *Let $\mathbb{G}_1$, $\mathbb{G}_2$ be well-formed terms generated by the design algebra. Then, $\mathbb{G}_1 \equiv_d \mathbb{G}_2$ if and only if $[\![\mathbb{G}_1]\!]$ is isomorphic to $[\![\mathbb{G}_2]\!]$.*

Moreover, the encoding is surjective.

Proposition 2 (cf. [5]). *Let G be a graph. Then, there exists a well-formed term $\mathbb{G}$ generated by the design algebra such that G is isomorphic to $[\![\mathbb{G}]\!]$.*

3.2 Side-view Model

The graphs-within-edges model corresponds, to some extent, to the top-view of the system. Another possibility is to take a side-view of the system, where containment is traced by dependencies between items in different layers (analogous to the representation of inheritance via arrows in UML class diagrams).

In [3] we have followed the side-view approach to interpret (a slight variation of) the algebra in § 2 over a class of graphs already available in the literature, called gs-graphs [14]. Roughly, gs-graphs are an extension of term-graphs [1] tailored to many-sorted hypersignatures. Moreover, in the formalisation of the model we have exploited the algebraic structure of gs-graph in terms of the so-called gs-monoidal theories [8]. Here we extend [3] to the design algebra of Def. 1 that allows for a more general form of interface.

While we refer the interested reader to [3] for most technical details, the main idea is to take a signature $\Sigma_{\mathcal{D},\mathcal{E}}$ whose sorts correspond to node sorts and whose operators correspond to the labels of edges. One additional sort $\bullet$ is also

$$\text{(op)}\ \frac{f \in \Sigma_{u,v}}{f : u \to v} \qquad \text{(id)}\ \frac{u \in S^*}{id_u : u \to u} \qquad \text{(bang)}\ \frac{u \in S^*}{!_u : u \to \epsilon} \qquad \text{(dup)}\ \frac{u \in S^*}{\nabla_u : u \to uu}$$

$$\text{(sym)}\ \frac{u, v \in S^*}{\rho_{u,v} : uv \to vu} \qquad \text{(seq)}\ \frac{t : u \to v \quad t' : v \to w}{t; t' : u \to w} \qquad \text{(par)}\ \frac{t : u \to v \quad t' : u' \to v'}{t \otimes t' : uu' \to vv'}$$

Fig. 3. Inference rules of gs-monoidal theories

introduced to represent "locations" within the hierarchy. Formally, for nodes sorted over S and edges labelled over $\mathcal{D} \cup \mathcal{E}$, we let $S^\bullet = S \cup \{\bullet\}$, assuming that $\bullet \notin S$, and let $\Sigma_{\mathcal{D},\mathcal{E}}$ denote the signature over $S^\bullet$ defined as follows:

$$\Sigma_{\mathcal{D},\mathcal{E}} = \{l \colon \bullet, ar(l) \to \epsilon \mid l \in \mathcal{E}\} \cup \{L \colon \bullet, ar(L) \to \bullet, ar(L) \mid L \in \mathcal{D}\} \cup \{\nu_s \colon \bullet \to s \mid s \in S\}$$

Thus, each hierarchical edge $L \in \mathcal{D}$ defines an operator $L \in \Sigma_{\mathcal{D},\mathcal{E}}$ that takes as arguments a location and the list of actual parameters and returns a location and the list of formal parameters (i.e., it provides the inner graph with the location where to reside and with a local environment). Of course, the type and number of parameters corresponds to the rank of L. Plain edges l provide no result (their co-arity is ϵ, the empty list)

By analogy with the well-known construction that given an ordinary signature allows to define its initial model as the free cartesian category of terms over that signature, starting from $\Sigma_{\mathcal{D},\mathcal{E}}$ we can generate the so-called free gs-monoidal theory $\mathbf{GS}(\Sigma_{\mathcal{D},\mathcal{E}})$, that accounts for all the gs-graphs that can be defined over the signature $\Sigma_{\mathcal{D},\mathcal{E}}$: differently from cartesian categories, gs-monoidal categories account for the sharing of sub- terms/graphs and for the presence of hidden sub-terms/graphs.

The expressions of interest are generated by the rules depicted in Fig. 3: they are obtained from some basic (families of) terms by closing them with respect to *sequential* (seq) and *parallel* (par) *composition*. By rule (op), the basic terms include one generator for each operator of the signature: these can be considered as the elementary bricks of our expressions, and conceptually correspond to the hyperedges of the term graphs. All other basic terms define the wires that can be used to build our graphs: the *identities* (id), the *dischargers* (bang), the *duplicators* (dup) and the *symmetries* (sym).

Note that expressions t are "typed" over pairs of lists of sorts and that their types determine the admissibility of sequential composition. For $t : u \to v$, with respect to our intuitive view of systems, the *source* u expresses the top-interface of t, that must be matched when embedding the expression in a larger context; the *target* v expresses the inner-interface, that constrains the admissible sub-graphs that can be placed below t; sequential composition represents the placing of one system (e.g. $t' : v \to w$) below another (e.g. $t; t'$).

Definition 9 (gs-monoidal theory). *Given a hypersignature Σ over a set of sorts S, the associated* gs-monoidal theory $\mathbf{GS}(\Sigma)$ *is the category whose objects* *are the elements of* S^*, *and whose arrows are equivalence classes of* gs-monoidal terms, *i.e., terms generated by the inference rules in Fig. 3 subject to the following conditions*

- *identities and sequential composition satisfy the axioms of categories:*
  [identity] $id_u \,;\, t = t = t \,;\, id_v$ *for all* $t : u \to v$*;*
  [associativity] $t_1 \,;\, (t_2 \,;\, t_3) = (t_1 \,;\, t_2) \,;\, t_3$ *whenever any side is defined,*
- $\otimes$ *is a monoidal functor with unit* id_ϵ*, i.e. it satisfies:*
  [functoriality] $id_{uv} = id_u \otimes id_v$*, and*
  $(t_1 \otimes t_2) \,;\, (t'_1 \otimes t'_2) = (t_1 \,;\, t'_1) \otimes (t_2 \,;\, t'_2)$ *whenever both sides are defined,*
  [monoid] $t \otimes id_\epsilon = t = id_\epsilon \otimes t \qquad t_1 \otimes (t_2 \otimes t_3) = (t_1 \otimes t_2) \otimes t_3$
- ρ *is a natural transformation, i.e. it satisfies:*
  [naturality] $(t \otimes t') \,;\, \rho_{v,v'} = \rho_{u,u'} \,;\, (t' \otimes t)$ *for all* $t : u \to v$ *and* $t' : u' \to v'$ *and furthermore it satisfies:*
  [symmetry] $(id_u \otimes \rho_{v,w}) \,;\, (\rho_{u,w} \otimes id_v) = \rho_{u \otimes v,w} \qquad \rho_{u,v} \,;\, \rho_{v,u} = id_{u \otimes v}$
  $\rho_{\epsilon,u} = \rho_{u,\epsilon} = id_u$
- ∇ *and* $!$ *satisfy the following axioms:*
  [unit] $!_\epsilon = \nabla_\epsilon = id_\epsilon$
  [duplication] $\nabla_u \,;\, (id_u \otimes \nabla_u) = \nabla_u \,;\, (\nabla_u \otimes id_u) \qquad \nabla_u \,;\, (id_u \otimes !_u) = id_u$
  $\nabla_u \,;\, \rho_{u,u} = \nabla_u$
  [monoidality] $\nabla_{uv} \,;\, (id_u \otimes \rho_{v,u} \otimes id_v) = \nabla_u \otimes \nabla_v \qquad !_{uv} = !_u \otimes !_v$

We call a *wiring* any arrow of $\mathbf{GS}(\Sigma)$ which is obtained from the rules of Fig. 3 without using rule (op). Notice that the definition of wiring is well-given, because any operator symbol introduced by rule (op) is preserved by all the axioms of the theory. Notably, the wirings of $\mathbf{GS}(\Sigma)$ from u to v are in bijective correspondence with the set of functions $\{k \colon \underline{|v|} \to \underline{|u|} \mid u[k(i)] = v[i] \text{ for all } 1 \leq i \leq |v|\}$, where for an ordinal $n \in \mathbb{N}$, we write $\underline{n}$ for the set $\{1, \ldots, n\}$.

The key consequence is that when drawing gs-graphs, we can abstract away from the way and order in which tentacles cross each other, because the axioms of gs-monoidal theories establish the equivalence of all drawings representing the same (set of) connections.

Then, each term $\mathbb{G}$ is translated to a gs-graph having $\bullet$ followed by (a linearisation of) the sorts of free nodes $fn(\mathbb{G})$ as source interface and the empty list of sorts ϵ as target interface. To fix the set-to-list correspondence between $fn(\mathbb{G})$ and the source interface, we exploit the concept of an *assignment*.

Definition 10 (Assignment). *An* assignment *is a function* $\sigma \in \bigcup_{n \in \mathbb{N}} \{f \colon \underline{n} \to \mathcal{X} \times S \mid f \text{ is injective}\}$*. An assignment* $\sigma \colon \underline{n} \to \mathcal{X} \times S$ *for a given* $n \in \mathbb{N}$ *is uniquely determined by a list of nodes without repetitions (because it is injective), namely* $\sigma(1), \sigma(2), \ldots, \sigma(n)$*: we shall often represent it this way and write* $x{:}s \in \sigma$ *as a shorthand for* $x : s$ *belonging to* $img(\sigma)$*, the image of* σ*.*

In the following, by $\tau(\sigma)$ we denote $\tau(\sigma(1), \sigma(2), \ldots, \sigma(n))$, i.e., the sequence of sorts of the nodes in $img(\sigma)$. Furthermore, for a given list of nodes $\overline{y} \in (\mathcal{X} \times S)^*$ and an assignment σ such that $|\overline{y}| \subseteq img(\sigma)$, we let $k^{\sigma}_{\overline{y}} \colon \underline{|\overline{y}|} \to \underline{|\sigma|}$ be the function such that $k^{\sigma}_{\overline{y}}(i) = \sigma^{-1}(\overline{y}|_i)$ for all $0 < i \leq |\overline{y}|$.

Definition 11 (GS-graph encoding). *Given an assignment* $\sigma = x_1{:}s_1, \ldots, x_n{:} s_n$ *and a term* $\mathbb{G}$ *with* $fn(\mathbb{G}) \subseteq img(\sigma)$ *such that all its bound variables carry different names*[5] *(also different from the names in* σ*), we define the gs-graph* $[\![\mathbb{G}]\!]_\sigma \colon \bullet, \tau(\sigma) \to \epsilon$ *by structural induction as follows (assuming that* $\otimes$ *has conventional precedence over* ;*):*

- $[\![\mathbf{0}]\!]_\sigma = [\![x : s]\!]_\sigma = !_{\bullet,\tau(\sigma)} : \bullet, \tau(\sigma) \to \epsilon$
- $[\![l\langle\overline{y}\rangle]\!]_\sigma = id_\bullet \otimes wir(k_{\overline{y}}^{\sigma})\ ;\ l : \bullet, \tau(\sigma) \to \epsilon$, *where the expression* $wir(k_{\overline{y}}^{\sigma}) : \tau(\sigma) \to ar(l)$ *is the wiring uniquely determined by* $k_{\overline{y}}^{\sigma} \colon \lfloor ar(b) \rfloor \to \lfloor \sigma \rfloor$.
- $[\![L_{\overline{x}}[\mathbb{G}]\langle\overline{y}\rangle]\!]_\sigma = id_\bullet \otimes \nabla_{\tau(\sigma)}\ ;\ (id_\bullet \otimes wir(k_{\overline{y}}^{\sigma})\ ;\ L) \otimes id_{\tau(\sigma)}\ ;\ [\![\mathbb{G}]\!]_{\overline{x},\sigma} : \bullet, \tau(\sigma) \to \epsilon$, *where w.l.o.g. we assume* $\lfloor \overline{x} \rfloor \cap \lfloor \sigma \rfloor = \emptyset$ *and the expression* $wir(k_{\overline{y}}^{\sigma}) : \tau(\sigma) \to ar(L)$ *is the wiring uniquely determined by* $k_{\overline{y}}^{\sigma} \colon \lfloor ar(b) \rfloor \to \lfloor \sigma \rfloor$.
- $[\![\mathbb{G}|\mathbb{H}]\!]_\sigma = \nabla_{\bullet,\tau(\sigma)}\ ;\ [\![\mathbb{G}]\!]_\sigma \otimes [\![\mathbb{H}]\!]_\sigma : \bullet, \tau(\sigma) \to \epsilon$
- $[\![(\nu\ x : s)\mathbb{G}]\!]_\sigma = [\![\mathbb{G}]\!]_\sigma : \bullet, \tau(\sigma) \to \epsilon$ *if* $x : s \notin fn(\mathbb{G})$
- $[\![(\nu\ x : s)\mathbb{G}]\!]_\sigma = (\nabla_\bullet\ ;\ id_\bullet \otimes \nu_s) \otimes id_{\tau(\sigma)}\ ;\ [\![\mathbb{G}]\!]_{x:s,\sigma} : \bullet, \tau(\sigma) \to \epsilon$ *otherwise, where w.l.o.g. we assume* $x : s \notin \lfloor \sigma \rfloor$.

Note that although $[\![\mathbf{0}]\!]_\sigma$ and $[\![x : s]\!]_\sigma$ are defined in the same way, the first is defined for any σ, while the second one is defined only if $x : s \in \sigma$.

Theorem 2 (cf. [3]). *Let* $\mathbb{G}$ *and* $\mathbb{H}$ *be two terms such that* $\mathbb{G} \equiv_{\mathrm{D}} \mathbb{H}$ *iff for any assignment* σ *we have* $[\![\mathbb{G}]\!]_\sigma = [\![\mathbb{H}]\!]_\sigma$.

Contrary to the encoding in § 3.1 the encoding applies to a restricted class of terms and is not surjective: The crucial fact is that the scoping discipline of restriction restricts the visibility of a localised nodes $x : s$ in such a way that it cannot be used from edges outside the one where $(\nu\ x : s)$ appears, but such a node scoping discipline has no counterpart in gs-graphs. This fact suggests that our algebra can serve to characterise exactly those term graphs with well-scoped references to nodes.

We conclude by sketching in Fig. 4 the gs-graphs corresponding to the two hierarchical graphs in Fig. 1: $A[(\nu w)(a\langle x, w\rangle \mid a\langle w, y\rangle)]\langle x, y\rangle$ on the left, and $(\nu w)(\, A_{u,v}[\mathbb{G}]\langle x, y\rangle \mid A_{u,v}[\mathbb{G}]\langle y, x\rangle\,)$ on the right (for $\mathbb{G} = a\langle u, w\rangle \mid a\langle w, v\rangle$). The drawing is decorated with: an external dashed line enclosing the gs-graph and emphasising its interface, the names of free nodes available, some dotted lines suggesting the correspondence between actual and formal parameters of A-labelled edges. Such a decoration is not part of the formal definition and has the only purpose to ease the intuitive correspondence with Fig. 1.

---

[5] This also means that in any occurrence of $L_{\overline{x}}[\mathbb{G}]$ the list $\overline{x}$ has no repetitions.

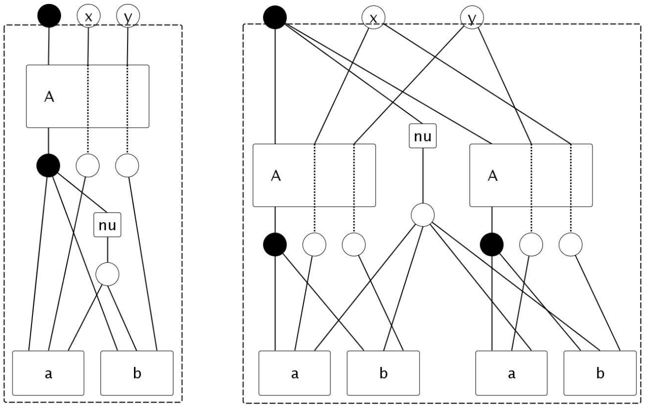

Fig. 4. Hierarchical structure as gs-monoidal terms

4 Applications to Service-Oriented Systems

This section presents one possible application of our approach, namely the graphical encoding of process calculi. We first discuss some methodological aspects and then show two examples, where the emphasis is respectively on the hierarchical nature of transactions and sessions.

4.1 Encoding Methodology

The main idea for defining graphical encoding of process calculi is to interpret process constructors as derived operators of our algebra. In that manner, each process term corresponds to a graph term, and hence to a hierarchical and gs-graph offering both a top and a side view of the same process. Moreover, if the interpretation faithfully captures the structural congruence of the calculus with the axioms of the graph algebra we obtain a nice result: congruent processes uniquely correspond to isomorphic graphs, and vice versa.

Each derived operator defines thus a graph operation that introduces items (nodes and edges). The first step of our methodology is fixing the set of node sorts, edge labels and design labels. Nodes are typically used to represent channels and control points and are sorted accordingly, while plain edges represent constructs such as atomic activities. Instead, inherently hierarchical constructors like session and transaction scopes are represented by designs.

Moreover, other design sorts can be introduced (i.e. one for each syntactical category of the calculus) to play the role of type annotations and constrain the applicability of derived operators, but they must be removed once the graphs are composed. For instance, parallel composition and non-deterministic choices are typically interpreted as graph operations that do not introduce any graph item, thus reflecting the axioms associated to such operations (associativity and commutativity).

The removal of such annotations is done by introducing flattening axioms, which implicitly remove (by performing some kind of hyper-edge replacement [12]) those edges satisfying a specific membership predicate (i.e. being typed with the annotation sorts).

Definition 12 (flattening axiom). *The flattening axiom* $\mathsf{flat}_{\mathsf{L}}$ *for a design label* L *is* $L_{\overline{x}}[\mathbb{G}]\langle\overline{y}\rangle \equiv \mathbb{G}\{^{\overline{y}}/_{\overline{x}}\}$.

It is evident that when $\mathsf{flat}_{\mathsf{L}}$ is considered, then L-labelled edges are immaterial. Flattening is fundamental in order to characterise classes of graphs by means of derived operators. Indeed, flattening is used in all encondings, where some design labels are used just for the sake of composing various classes of processes and not really to build scopes. So nesting has two roles: as a means to enclose a graph and as a sort of typed interface to enable disciplined graph compositions. The presence of flattening axioms makes the first role immaterial.

Another kind of axioms that are sometimes useful to be included in the structural congruence are *extrusion* axioms.

Definition 13 (extrusion axiom). *The extrusion axiom* $\mathsf{extr}_{\mathsf{L}}$ *for a design label* L *is* $L_{\overline{x}}[(\nu z)\mathbb{G}\langle\overline{y}\rangle] \equiv (\nu z)L_{\overline{x}}[\mathbb{G}]\langle\overline{y}\rangle$*, where* $z \notin \lfloor\overline{x}\rfloor \cup \lfloor\overline{y}\rfloor$.

Extrusion axioms are needed to handle those calculi in which name restriction is not localised inside a scope or it is allowed to cross the boundaries of some scopes, as it may happen for some process calculi. Indeed, we shall see in § 4.3 how extrusion axioms are used to capture extrusion for some scope constructs.

Note that the addition of axiom $\mathsf{flat}_{\mathsf{L}}$ also implies the validity of axiom $\mathsf{extr}_{\mathsf{L}}$, hence in the following we assume that for each label L exactly one of the following cases applies: either only the extrusion or only the flattening axiom for L is present; or none of $\mathsf{flat}_{\mathsf{L}}$ and $\mathsf{extr}_{\mathsf{L}}$ is present. Of course the presence of such axioms for a chosen set of labels is often fundamental for the soundness of the encoding.

4.2 Transaction Workflows

We consider in this section the *nested sagas with programmable compensations* of [6], a calculus for long running transactions that aims at providing a core language for composing activities into *sagas* (atomic transactions) or *processes* (non-atomic compensable activities). Formally, the syntax of sagas is as follows.

Definition 14 (sagas syntax). *Let* Λ *be a set of atomic activities ranged over by* a*. The sets* $\mathcal{S}$ *of sagas and* $\mathcal{P}$ *of compensable processes are all the terms generated by* S *and* P *in the grammar below, respectively.*

$$S \quad ::= a \;\mid\; \{P\} \qquad\qquad P \quad ::= S\%S \;\mid\; P;P \;\mid\; P \mid P$$

For the sake of simplicity, with respect to the original presentation we neglect the introduction of nil processes and non-compensable activities. A *saga*

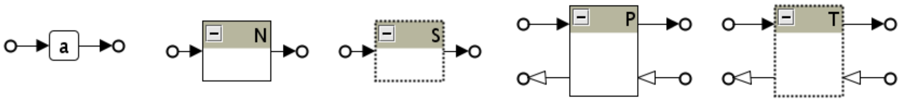

Fig. 5. Type graph for sagas

is an atomic activity or an arbitrarily complex transaction built out from a compensable processes. A basic process $A\%B$ is built by declaring a saga A as an ordinary flow and equipping it with another saga B as its compensation flow. The calculus provides also primitives for composing processes in sequence and parallel (split&join).

Definition 15 (sagas structural congruence). *The* structural congruence *for sagas is the relation $\equiv_{\mathsf{S}} \subseteq \mathcal{P} \times \mathcal{P}$, closed under sagas construction, inductively generated by the following set of axioms (for any $P, Q, R \in \mathcal{P}$):*

$$\begin{array}{rcll} P;(Q;R) & \equiv & (P;Q);R & (s\mathsf{A1}) \\ P \mid Q & \equiv & Q \mid P & (s\mathsf{A2}) \\ P \mid (Q \mid R) & \equiv & (P \mid Q) \mid R & (s\mathsf{A3}) \end{array}$$

Encoding sagas. We now define the graphical encoding of sagas. As explained, the first step is to interpret syntactical categories of the calculus as suitable design labels and constructors as derived operators over our graph algebra. In this case we decide to introduce design labels N for Nested sagas, S for Sagas, P for compensable Pairs and T (Transactions) for compensable processes. Note that N can be read as a subsort of S, while P as a subsort of T. Figure 5 illustrates the shapes of the nodes and boxes we shall exploit. We have chosen an arity of four tentacles for pairs and transactions to denote the following control points: entry of the ordinary flow (incoming filled arrow), exit of the ordinary flow (outgoing filled arrow), entry of the compensation flow (incoming empty arrow) and exit of the compensation flow (outgoing empty arrow). Activities and sagas are represented by edges with only two tentacles (for the ordinary flow). Note that we have actually a family of activity edges, one for each activity in Λ. Since S and T are just used for composition, we let the flattening axioms $\mathsf{flat}_{\mathsf{S}}$ and $\mathsf{flat}_{\mathsf{T}}$ hold (whence the dotted borders in Fig. 5).

The encoding is formally defined as follows (cf. Fig. 6).

Definition 16 (sagas encoding). *The interpretation of the sagas operators over the design algebra is given by*

$$\begin{array}{rcl} a & \stackrel{\mathbf{def}}{=} & S_{p,q}[a\langle p,q\rangle] \\ \{Q\} & \stackrel{\mathbf{def}}{=} & N_{p,q}[(\nu t)Q\langle p,q,t,q\rangle] \\ A \ \% \ B & \stackrel{\mathbf{def}}{=} & P_{p,q,r,s}[A\langle p,q\rangle \mid B\langle r,s\rangle] \\ Q \ ; \ R & \stackrel{\mathbf{def}}{=} & T_{p,q,r,s}[(\nu u,v)(Q\langle p,u,v,s\rangle \mid R\langle u,q,r,v\rangle)] \\ Q \ \mid \ R & \stackrel{\mathbf{def}}{=} & T_{p,q,r,s}[Q\langle p,q,r,s\rangle \mid R\langle p,q,r,s\rangle] \end{array}$$

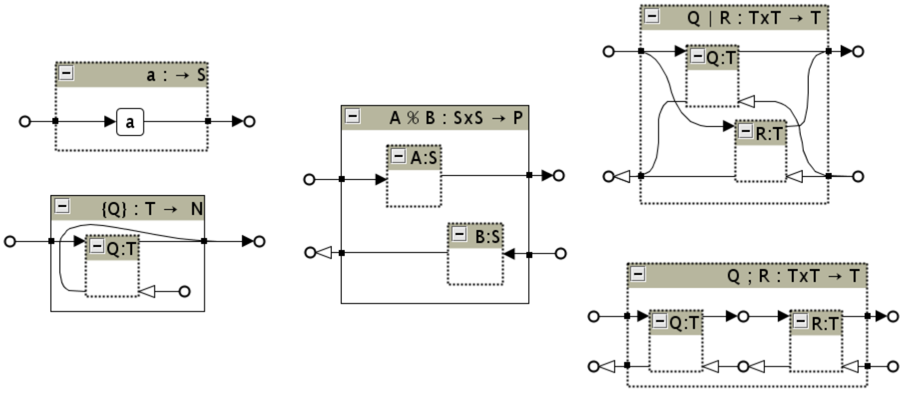

Fig. 6. Graphical interpretation for sagas.

Note again that some primitives of the calculus are considered as material in the encoding, i.e. represented by graph items like edges. This is the case of activities as shown in Fig. 5 and also of compensable pairs and nested sagas. Instead, sequencing and parallel composition (see Fig. 6) are immaterial and their associated axioms are captured by the flattening axioms.

The proposed encoding is sound and complete, i.e. equivalent processes and sagas are mapped into isomorphic graphs as shown in [5].

Proposition 3 (cf. [5]). *For any* $Q, R \in \mathcal{P}$ *we have* $Q \equiv_{\mathrm{S}} R$ *iff* $Q \equiv_{\mathrm{D}} R$.

Example 3. Consider the following example, inspired from [6] of the saga

{acceptOrder%refuseOrder ; (updateCredit%refundOrder |
prepareOrder%updateStock) |
{addPoints%skip}%{substractPoints%skip}) }

The saga is used for modelling a scenario for dealing with purchase orders. The initial activity (*acceptOrder*) handles requests from clients. The next three processes are executed in parallel. The first one (*updateCredit*) charges the amount of the order to the balance of the client. The second one (*prepareOrder*) handles the packaging of the order and updates the stock. The third one deals with point reward activities: it is formed by a nested saga to update the reward balance of a user (part of a program for accumulating points with purchases). All the activities have a corresponding compensation to undo the actions performed by the successful completion of the activities. Note that activity *addPoints* has a vacuous compensation (*skip*) to avoid aborting the purchase when the point accumulation activity aborts due to the absence of a reward account (idem for activity *substractPoints*). The corresponding hierarchical graph is in Fig. 7.

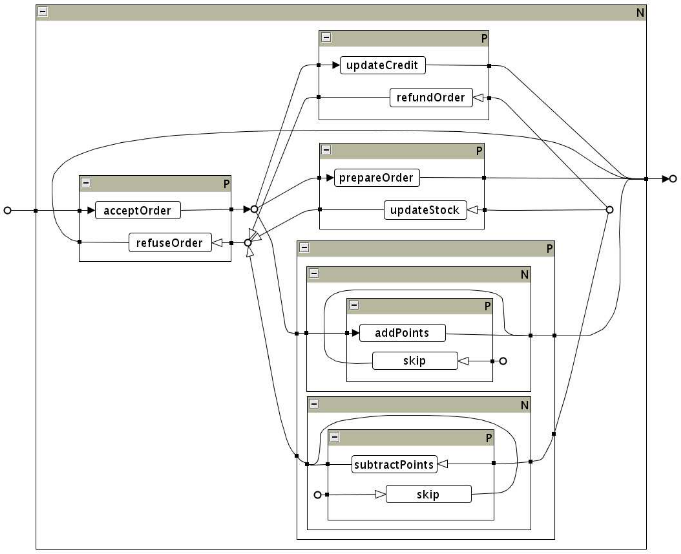

Fig. 7. Graphical encoding of a saga

4.3 Service Sessions

This section sketches the graphical representation of CaSPiS [2], a session-centred calculus developed within SENSORIA. We have chosen this calculus since it represents a non-trivial example of the interplay between nesting and linking introduced by nested sessions, pipelines and communication. We briefly overview CaSPiS and we refer the interested readers to [2] for an exhaustive description. We remark that we focus here on the close-free fragment of the calculus and we present a slightly simplified syntax. Both decisions are for the sake of a convenient and clean presentation and constitute no limitation.

Definition 17 (CaSPiS syntax). *Let* $\mathcal{Z}$ *be a set of session names,* $\mathcal{S}$ *a set of service names and* $\mathcal{V}$ *a set of value names. The set* $\mathcal{P}$ *of processes consists of all the terms generated by* P *in the grammar below*

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

where $s \in \mathcal{S}$, $r \in \mathcal{Z}$, $u \in \mathcal{V}$, $w \in \mathcal{V} \cup \mathcal{Z}$ *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 $r \triangleright P$, and $r \triangleright Q$, 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 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.

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 sides, or used locally to invoke other services, to start new activities. Local consumption 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 runs within the same session as $P > Q$, not in a fresh one.

Summarising, each CaSPiS process can be thought of as running in an environment providing it different means of communication: one channel for "standard" input, one channel for "standard" output and one channel for returning values one level up.

Example 4. Consider the process $(\nu a)(\nu b)(a \triangleright (P_1 | b \triangleright P_2) | a \triangleright P_3 | b \triangleright P_4)$. It represents a typical situation where two sessions a and b have been created (e.g. upon 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, and they run separately. Notably, values returned one level up by P_2 can be consumed by P_3.

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

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

Encoding CaSPiS. We first define the alphabets of edge labels and nodes. The set $\mathcal{D}$ 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). The set $\mathcal{E}$ 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 corresponding channel node.

The graphical representation of each design and edge label and their respective ranks can be found in Fig. 8. For instance, designs of type P are all of the form $P_{p,t,o,i}[\mathbb{G}]$ where p is the control point representing the process start of execution, t is the returning channel, o is the output channel and i is the input channel. Vice

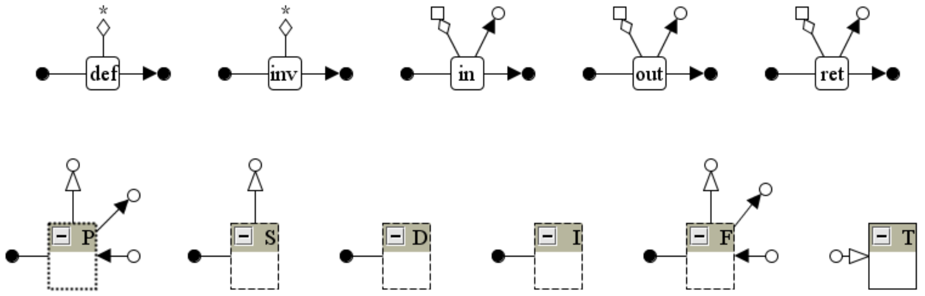

Fig. 8. Type graph for CaSPiS.

versa, designs of type D and I only expose the starting point of execution: they are not strictly necessary for the encoding, but can be very useful for visualisation purposes (they enclose the interaction protocols between callers and callees). We let the flattening axiom flat_P hold, together with extrusion axioms extr_S, extr_D, extr_I, extr_F. Hence, edges of type P are immaterial (they can be considered as type annotations) and edges of type T define the only *rigid* hierarchy w.r.t. containment and name scoping. Other explicit hierarchies for edge containment are given by session nesting (S), service definition (D), service invocation (I) and pipelining (F). As usual, flattening processes allows for getting rid of the axioms for parallel composition (see [15]). The presence of extrusion axioms is motivated by the structural congruence axioms of CaSPiS, namely $\mathsf{CA7}$ motivates extr_F, $\mathsf{CA8}$ motivates both extr_D and extr_I, and $\mathsf{CA9}$ motivates extr_S. Note that we use dashed border for designs for which the extrusion axiom hold, while designs to be flattened are depicted with dotted borders.

Definition 19 (CaSPiS encoding). *The interpretation of CaSPiS operators over the design algebra is given by*

$$
\begin{array}{rl}
s.Q \stackrel{\text{def}}{=} & P_{p,t,o,i}[\, t|o|i|\, D[\,(\nu q, t', o', i')(\mathsf{def}\langle p, s, q\rangle | Q\langle q, t', o', i'\rangle)\,]\langle p\rangle\,] \\
\overline{s}.Q \stackrel{\text{def}}{=} & P_{p,t,o,i}[\, t|o|i|\, I[\,(\nu q, t', o', i')(\mathsf{inv}\langle p, s, q\rangle | Q\langle q, t', o', i'\rangle)\,]\langle p\rangle\,] \\
r \rhd Q \stackrel{\text{def}}{=} & P_{p,t,o,i}[\, t|i|\, S[\, Q\langle p, o, r, r\rangle\,]\langle p, o\rangle\,] \\
Q > R \stackrel{\text{def}}{=} & P_{p,t,o,i}[\, o \mid (\nu m)(\, F[\, Q\langle p, t, m, i\rangle\,]\langle p, t, m, i\rangle \mid \\
 & \qquad\qquad\qquad T[\,(\nu q, t', o')R\langle q, t', o', m\rangle\,]\langle m\rangle\,)\,] \\
Q|R \stackrel{\text{def}}{=} & P_{p,t,o,i}[\, Q\langle p, t, o, i\rangle | R\langle p, t, o, i\rangle\,] \\
(\nu w)Q \stackrel{\text{def}}{=} & P_{p,t,o,i}[(\nu w)Q\langle p, t, o, i\rangle] \\
\mathbf{0} \stackrel{\text{def}}{=} & P_{p,t,o,i}[p|t|o|i] \\
\langle u\rangle.Q \stackrel{\text{def}}{=} & P_{p,t,o,i}[\,(\nu q)(\mathsf{out}\langle p, q, u, o\rangle \mid Q\langle q, t, o, i\rangle)\,] \\
\langle u\rangle^{\uparrow}.Q \stackrel{\text{def}}{=} & P_{p,t,o,i}[\,(\nu q)(\mathsf{ret}\langle p, q, u, t\rangle \mid Q\langle q, t, o, i\rangle)\,] \\
(?x).Q \stackrel{\text{def}}{=} & P_{p,t,o,i}[\,(\nu q, x)(\mathsf{in}\langle p, q, x, i\rangle \mid Q\langle q, t, o, i\rangle)\,]
\end{array}
$$

Proposition 4 (cf. [5]). *For any $Q, R \in \mathcal{P}$ we have $Q \equiv_{\mathrm{C}} R$ iff $Q \equiv_{\mathrm{D}} R$.*

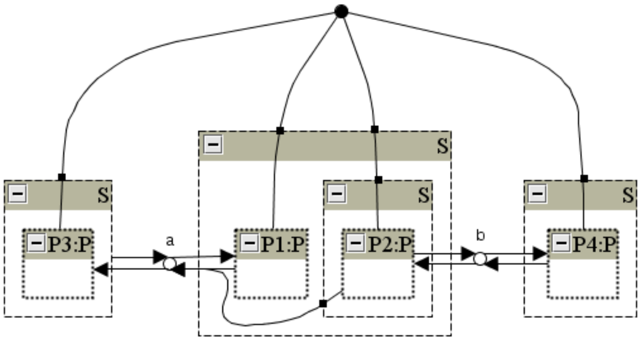

Fig. 9. Example of session nesting.

Instead of providing the visualisation of the encoding and a detailed explanation (for which we refer to [4]) we prefer to concentrate on the representation of session nesting with the typical session situation presented before. Figure 9 depicts the graphical representation of our example, where the graph has been further simplified (e.g. fusing nodes, removing isolated nodes and irrelevant tentacles) to focus on the main issues and make immediate the correspondence with the process term. The figure evidences the hierarchy introduced by session nesting and how it is crossed by intra-session communication. It is also worth to note that the graph highlights the fact that the return channel of a nested session is pipelined into the output channel of the enclosing session. More precisely, the return channel of the immediate session where P_2 lives (i.e. b) is connected to the output channel of the session containing it, i.e. the session channel a.

5 Conclusion

This chapter collects results from [3, 5, 4]. We presented our specification formalism based on a convenient algebra of hierarchical graphs: its features make it well-suited for the specification of systems with inherently hierarchical aspects and in particular, process calculi with notions of scope and containment (like ambients, membranes, sessions and transactions). Some advantages of our approach are due to the graph algebra, whose syntax resembles standard algebraic specifications and, in particular, it is close to the syntax found in nominal calculi. The key point is to exploit the algebraic structure of both designs and graphs when proving properties of an encoding, facilitating proofs by structural induction.

References

1. H. Barendregt, M. van Eekelen, J. Glauert, J. Kennaway, M. Plasmeijer, and M. Sleep. Term graph reduction. In *PARLE'87*, volume 259 of *Lecture Notes in Computer Science*, pages 141–158. Springer Verlag, 1987.

2. M. Boreale, R. Bruni, R. De Nicola, and M. Loreti. Sessions and pipelines for structured service programming. In G. Barthe and F. S. de Boer, editors, *Proceedings of the 10th IFIP International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS'08)*, volume 5051 of *Lecture Notes in Computer Science*, pages 19–38. Springer Verlag, 2008.
3. R. Bruni, A. Corradini, F. Gadducci, A. Lluch Lafuente, and U. Montanari. On gs-monoidal theories for graphs with nesting. In *Festschrift in Honour of Manfred Nagl 65th Birthday*, Lecture Notes in Computer Science. Springer Verlag, 2010. To appear.
4. R. Bruni, F. Gadducci, and A. Lluch Lafuente. A graph syntax for processes and services. In S. Jianwen and C. Laneve, editors, *WS-FM'09*, volume 6194 of *Lecture Notes in Computer Science*, pages 46–60. Springer Verlag, 2009. To Appear.
5. R. Bruni, F. Gadducci, and A. Lluch Lafuente. An algebra of hierarchical graphs and its application to structural encoding. *Scientific Annals of Computer Science*, 2010. To appear.
6. R. Bruni, H. C. Melgratti, and U. Montanari. Theoretical foundations for compensations in flow composition languages. In J. Palsberg and M. Abadi, editors, *Proceedings of the 32nd International Symposium on Principles of Programming Languages (POPL'05)*, pages 209–220. ACM, 2005.
7. M. Bundgaard and V. Sassone. Typed polyadic pi-calculus in bigraphs. In A. Bossi and M. J. Maher, editors, *Proceedings of the 8th International Symposium on Principles and Practice of Declarative Programming (PPDP'06)*, pages 1–12. ACM, 2006.
8. A. Corradini and F. Gadducci. An algebraic presentation of term graphs, via gs-monoidal categories. *Applied Categorical Structures*, 7:299–331, 1999.
9. A. Corradini, U. Montanari, and F. Rossi. An abstract machine for concurrent modular systems: CHARM. *Theoretical Computer Science*, 122(1-2):165–200, 1994.
10. 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 G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation*, pages 163–246. World Scientific, 1997.
11. F. Drewes, B. Hoffmann, and D. Plump. Hierarchical graph transformation. *Journal on Computer and System Sciences*, 64(2):249–283, 2002.
12. 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.
13. G. L. Ferrari, D. Hirsch, I. Lanese, U. Montanari, and E. Tuosto. Synchronised hyperedge replacement as a model for service-oriented computing. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, editors, *Proceedings of the 4th International Symposium on Formal Methods for Components and Objects (FMCO'05)*, volume 4111 of *LNCS*, pages 22–43. Springer, 2006.
14. G. L. Ferrari and U. Montanari. Tile formats for located and mobile systems. *Information and Computation*, 156(1-2):173–235, 2000.
15. F. Gadducci. Term graph rewriting for the pi-calculus. In A. Ohori, editor, *Proceedings of the 1st Asian Symposium on Programming Languages and Systems (APLAS'03)*, volume 2895 of *Lecture Notes in Computer Science*, pages 37–54. Springer Verlag, 2003.
16. R. Milner. Pure bigraphs: Structure and dynamics. *Information and Computation*, 204(1):60–122, 2006.