Scientific Annals of Computer Science vol. 20, 2010
"Alexandru Ioan Cuza" University of Iaşi, Romania

An Algebra of Hierarchical Graphs and its Application to Structural Encoding

Roberto BRUNI[1], Fabio GADDUCCI[1], Alberto LLUCH LAFUENTE[2]

Abstract

We define an algebraic theory of hierarchical graphs, whose axioms characterise graph isomorphism: two terms are equated exactly when they represent the same graph. Our algebra can be understood as a high-level language for describing graphs with a node-sharing, embedding structure, and it is then well suited for defining graphical representations of software models where nesting and linking are key aspects. In particular, we propose the use of our graph formalism as a convenient way to describe configurations in process calculi equipped with inherently hierarchical features such as sessions, locations, transactions, membranes or ambients. The graph syntax can be seen as an intermediate representation language, that facilitates the encodings of algebraic specifications, since it provides primitives for nesting, name restriction and parallel composition. In addition, proving soundness and correctness of an encoding (i.e. proving that structurally equivalent processes are mapped to isomorphic graphs) becomes easier as it can be done by induction over the graph syntax.

[1]Department of Computer Science, University of Pisa, Largo Bruno Pontecorvo 3, 56127 Pisa, Italy, email: bruni,gadducci@di.unipi.it
[2]IMT Institute for Advanced Studies Lucca, Piazza San Ponziano 6, 55100 Lucca, Italy, email: alberto.lluch@imtlucca.it

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 [18], Bigraphical Reactive Systems [23], Synchronized Hyperedge Replacement [17] and Membrane Systems [25, 26], just to cite a few of the most prominent examples.

Despite their expressiveness and flexibility, the use of these formalisms to build a graphical representation for an existing specification language involves the major challenge of encoding system configurations (states), guaranteeing that structural equivalence is preserved: any two equivalent configurations P and Q are mapped into isomorphic graphs $[\![P]\!]$ and $[\![Q]\!]$. Preserving structural equivalence has several advantages. It offers an intuitive normal form representation for systems, and it allows to reuse results and techniques from graph theory for solving specific problems: for example, checking structural equivalence by the use of algorithms for testing graph isomorphism. In particular, the soundness of the encoding is necessary to use graph transformation approaches [13] to model dynamic aspects like operational semantics, reconfigurations, refactorings or model transformations since (sub)graph isomorphism is at the base of the rule matching mechanism.

When configurations P are specified by using an algebraic syntax (e.g. as in process calculi), their encoding $[\![P]\!]$ can be driven by their (term) structure by defining it inductively. In the absence of an algebraic presentation for the language under consideration, an ad-hoc algebraic syntax must be developed if one wants to benefit from compositionality and structural induction in proofs, transformations or definitions. Still, most graph models are defined set-theoretically: most often, they are not equipped with a natural algebraic syntax and the existing ones require advanced skills to deal with sophisticated models involving ad-hoc definitions of graphs with interfaces (e.g. [18]) or complex type systems (e.g. [9]), or representing hierarchies as trees (e.g. [19, 23]), hampering definitions and proofs. Moreover, one encounters a severe drawback: namely, the syntax of those graph formalisms are often very different from the source language and they are 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. Summarising all the above, the representation

distance from the syntax of configurations with respect to the syntax of encoding graphs complicates *i*) the definition of the encoding; *ii*) its proof of correctness; and *iii*) its reuse when slightly different algebras of configurations or kinds of graphs are considered.

Our idea is to distill a sort of standard intermediate language between those used for specifying system configurations and those available for graph formalisms, where some essential first-class concepts are suitably represented and built-in so that *a*) a standard encoding from the intermediate syntax to the graph models and its correctness are established once and for all; *b*) the representation distance from system configurations to the intermediate syntax is considerably reduced; and *c*) the encoding of system configurations is then factorised via the intermediate language. The main advantage is that the definition of the encoding (and the proof of its correctness) are carried out more conveniently at the algebraic level.

We have decided to base our intermediate language on two key structural aspects that are arise repeatedly in system specifications, namely *nesting* and *linking*. Consider for instance the structure of file systems, composite diagrams, networks, membranes, sessions, transactions, locations, structured state machines or XML files. Various graphical models of nesting and sharing structures already exist but (as we claim in § 9) none of them offer a simple, intuitive syntax. Identifying the right structure is fundamental to enjoy scalability. In particular, nesting plays a fundamental role for abstracting the complexity of a system by offering different views at different levels of detail, based on the nesting depth.

This paper describes our proposal for addressing those challenges. The work presented here is the full version of two conference papers [5, 4], extended with the proofs of the main results and two original encodings. Below we clarify the sources of content in better detail while explaining the structure of the paper.

In [4], we have introduced a formalism made of an algebra (§ 2) for a model of hierarchical graphs (§ 3) to fill the gap between the different levels of abstraction at which algebraic specifications of software systems and graphical structures reside. The algebra enjoys primitives for dealing with names, restriction, parallel composition and, most importantly, nesting in the same way as they are used in process calculi. In particular, the nesting mechanism allows for easily defining graphical presentations of inherently hierarchical aspects such as locations, membranes, ambients, transactions or sessions, and it is equipped with a sound and complete set of axioms

equating two terms whenever they represent isomorphic graphs (§ 4). Besides facilitating the visual specification of configurations, we argue that definitions, transformations and proofs by induction are made easier by the algebraic structure of configurations and graphs.

In [5] we have validated the above idea by using our graph algebra to encode the configurations of two (process) calculi with service-inherent features that have a certain hierarchical nature such as sessions, transactions or locations: the first one is a simple workflow language, vaguely reminiscent of BPEL; the second example concerns a sophisticated calculus for the description of service-oriented applications, namely, CaSPiS [1] (see § 7), whose features pose further challenges to visualisation, due to the interplay of name handling, nested sessions and a pipeline operator. This paper extends [5] with the proof of the correspondence result for CaSPiS (§ 7), preceded by two novel encodings: § 5 shows the encoding of the best-known nominal calculus, namely the π-calculus [22] and § 6 focuses on a calculus for transactions called *sagas* [8]. Each example illustrates the treatment of linking, nesting and their combination, respectively. We remark that the technique we propose can be transferred to other calculi as well, as witnessed by other available encodings mentioned in § 9.

2 An algebra of hierarchical graphs

We introduce here our algebra of (typed) hierarchical graphs that we call *designs*. The algebraic presentation of designs has emerged during our studies on *Architectural Design Rewriting* [7] (hence the name) and has been inspired by the graph algebra of CHARM [12].

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

$$\begin{array}{lll} \mathbb{D} & ::= & L_{\overline{x}}[\mathbb{G}] \\ \mathbb{G} & ::= & \mathbf{0} \mid x \mid l\langle\overline{x}\rangle \mid \mathbb{G} \mid \mathbb{G} \mid (\nu x)\mathbb{G} \mid \mathbb{D}\langle\overline{x}\rangle \end{array}$$

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

The algebraic reading is as usual, where each syntactical category and vocabulary is considered as a sort and productions are read as functions. This allows us, for instance, to consider open terms (i.e. terms with typed variables): they are useful for defining encodings by means of derived operators, as we shall see in § 5, § 6 and § 7.

As a matter of notation, we let $\lfloor \overline{x} \rfloor$ denote the set of elements of a list $\overline{x}$; we also overload $|\cdot|$ in order to let it denote either the length of a list or the cardinality of a set.

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-labeled (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 labeled 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 length 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).

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

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

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

The following example offers a first intuition of the algebra and the model of hierarchical graphs. For this purpose we offer an informal, appealing visual notation. The formal underlying graphs are introduced in § 3.

Example 1. For simplicity, in this example we consider hyperedges that have two tentacles each, but this is not a restriction we shall enforce and in fact we will consider more general cases in the rest of the paper. Let $a, b \in \mathcal{E}$,

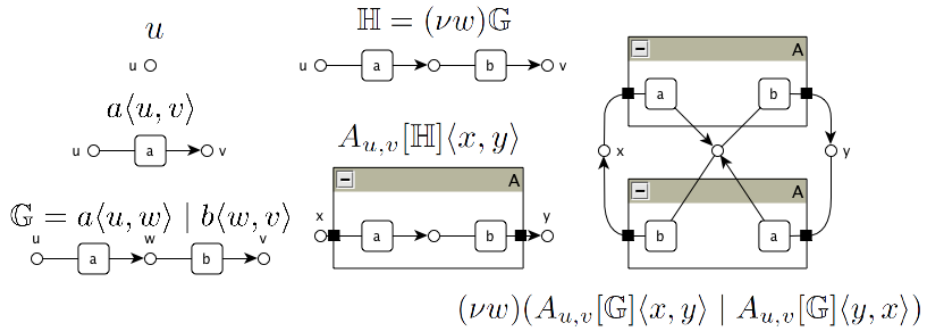

Figure 1: Some terms of the graph algebra

$A \in \mathcal{D}$, $u, v, w, x, y \in \mathcal{N}$. We write and depict in Figure 1 some terms of our algebra. 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. If a node is exposed in the interface we put it on the outermost layer and overlap the edges of the various layers denoting this with black boxes on design borders. In the particular examples only free nodes are annotated with their identities. Note that this representation is informal to give a first intuition of our model of hierarchical graphs. Next section offers the formal representation of the rightmost term.

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 topology. To this aim it is typically the case that: 1) nodes are sorted, in which case their labels take the form $n : s$ for n the *name* and s the *sort* of the node; 2) each label of $\mathcal{E}$ and $\mathcal{D}$ has a fixed arity and for each rank a fixed node sort; 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 $\mathbb{G}$). When this is the case, we say that a design (or a graph) is *well-typed* if for each sub-term $L_{\overline{x}}[\mathbb{G}]$ we have that the (lists of) sorts of $\overline{x}$ and L coincide, and similarly for sub-terms $\mathbb{D}\langle\overline{x}\rangle$ and $l\langle\overline{x}\rangle$. From now on, we restrict our attention to well-formed designs.

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

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

Intuitively, the restriction on the mapping $^{\overline{y}}/_{\overline{x}}$ allows $\overline{x}$ to account for matching and mismatching of nodes in the interface: distinct nodes in $\overline{y}$ must correspond to distinct nodes in $\overline{x}$, and if the list $\overline{x}$ contain repetitions, then all the occurrences of the same node x in $\overline{x}$ must correspond to the same node y in $\overline{y}$, and vice versa.

In order to have a notion of syntactically equivalent designs (i.e. to consider designs up to isomorphism), the algebra includes the structural graph axioms of [12] such as associativity and commutativity for $\mid$ (with identity $\mathbf{0}$) and node restriction (respectively, axioms DA1–DA3 and DA4–DA6). In addition, it includes axioms to α-rename bound nodes (DA7–DA8), an axiom for making immaterial the addition of a node x to a graph where x is already free (DA9) and another one that makes sure global names are not localized inside designs (DA10).

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

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

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

Note that $\equiv_{\mathrm{D}}$ respects free nodes: $\mathbb{G} \equiv_{\mathrm{D}} \mathbb{H}$ implies $fn(\mathbb{G}) = fn(\mathbb{H})$. Being $\equiv_{\mathrm{D}}$ a congruence, we remark 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 that it allows the derivation of standard representatives for the equivalence classes induced by $\equiv_{\mathrm{D}}$.

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

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

where all terms $\mathbb{G}_i$ are themselves in normalized 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 $X \subseteq Z$, $fn(\mathbb{G}) = Z \setminus X$ and $fn(L^i_{\overline{y}_i}[\mathbb{G}_i]) = Z$ for all $i = 1...q$.

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

Proof: We proceed by structural induction. For the base cases we have that $\mathbf{0}$ and x already in normalized form and the single-edged graph $l\langle\overline{x}\rangle$ can be put in normalized form by exploiting DA9 to add in parallel composition the nodes in $\lfloor\overline{x}\rfloor$. For the inductive cases, assume $norm(\mathbb{G})$ and $norm(\mathbb{H})$ are the normalized forms of $\mathbb{G}$ and $\mathbb{H}$ respectively. The normalized form of the graph $(\nu x)\mathbb{G}$ is $(\nu x)norm(\mathbb{G})$ if $x \in fn(\mathbb{G})$, otherwise it is just $norm(\mathbb{G})$, because $(\nu x)\mathbb{G} \equiv_{\mathrm{D}} (\nu x)(\mathbb{G} \mid \mathbf{0}) \equiv_{\mathrm{D}} \mathbb{G} \mid (\nu x)\mathbf{0} \equiv_{\mathrm{D}} \mathbb{G} \mid \mathbf{0} \equiv_{\mathrm{D}} \mathbb{G}$ (by axioms DA3, DA6 and DA5). For the last two cases, we introduce the notation $normsat(\mathbb{G}, Y)$, where Y is a set of nodes, to denote the term obtained by saturating $norm(\mathbb{G})$ by the parallel composition of the nodes in Y. This can be straightforwardly defined by: 1) alpha-converting all the bound names appearing in $norm(\mathbb{G})$ so to be all different from the names in Y (this is achieved by axioms DA7 and DA8); 2) put $\mathbb{G}$ in parallel with $\mathbb{Y}$, where $\mathbb{Y}$ is the parallel composition of all names in $Y \setminus fn(\mathbb{G})$; 3) exploit axiom DA9 to create as many duplicates of $\mathbb{Y}$ as the top-level design edges appearing in $norm(\mathbb{G})$; 4) exploit axioms DA1, DA2 and DA6 to place one copy of $\mathbb{Y}$ right after the top-level restrictions and the other copies of $\mathbb{Y}$ nearby each top-level design edge; 5) exploit axiom DA10 to move the copies of $\mathbb{Y}$ inside their adjacent design edges and then apply iteratively this procedure (from step 2) to their content. Then, the normalized form of $L_{\overline{x}}[\mathbb{G}]\langle\overline{y}\rangle$ is $\mathbb{Z} \mid L_{\overline{x}}[normsat(\mathbb{G}, Y)]\langle\overline{y}\rangle$, where $Y = \lfloor\overline{y}\rfloor \setminus fn(L_{\overline{x}}[\mathbb{G}])$ and $\mathbb{Z}$ is the parallel composition of nodes in $\lfloor\overline{y}\rfloor \cup fn(L_{\overline{x}}[\mathbb{G}])$ (the equivalence is proved by

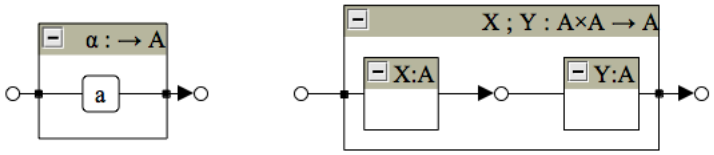

Figure 2: An algebra of graph sequences.

applying axioms DA9 and DA10 repeatedly). Finally, the normalized form of $\mathbb{G} \mid \mathbb{H}$ is obtained by: 1) taking $normsat(\mathbb{G}, Y) \mid normsat(\mathbb{H}, Z)$, where $Y = fn(\mathbb{H}) \setminus fn(\mathbb{G})$ and $Z = fn(\mathbb{G}) \setminus fn(\mathbb{H})$; 2) alpha-converting all the top-level restricted names appearing in $normsat(\mathbb{H}, Z)$ so to be all different from the bound names in $normsat(\mathbb{G}, Y)$ and then group all such top-level restrictions to the left; 3) rearrange the term using axioms DA1 and DA2 to group similar items (nodes, edges and design edges); 4) exploit axiom DA9 to remove duplicate top-level nodes. □

Roughly, in $norm(\mathbb{G})$ 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 normalized form can be proved to be unique.

We call a graph *flat* whenever there is no design in its body. Sometimes, we impose the flattening property on the graph algebra by an axiom schema, implicitly removing (by performing some kind of hyper-edge replacement [15]) those edges satisfying a specific membership predicate.

Definition 6 (flattening axiom) *A flattening axiom* $\mathsf{flat}_{\mathsf{L}}$ *for some design label* L *is of the form* $L_{\overline{x}}[\mathbb{G}]\langle\overline{y}\rangle \equiv \mathbb{G}\{^{\overline{y}}/_{\overline{x}}\}$.

In the next example we see how flattening is fundamental in order to characterise classes of graphs by means of derived operators. Indeed, flattening is used in all three encoding examples (see § 5, 6 and § 7) where some design labels will be used just for the sake of composing various classes of processes and not really to build scopes.

Example 2. Suppose that we want to characterise the set A of a-labelled, acyclic, and connected sequences (see Example 1). We can define an algebra with an element $\alpha :\rightarrow A$ in the sequence, and a binary sequential composition $\_;\_ : A \times A \rightarrow A$. Both are derived operators defined by $\alpha \stackrel{\mathbf{def}}{=} A_{u,v}[a(u,v)]$

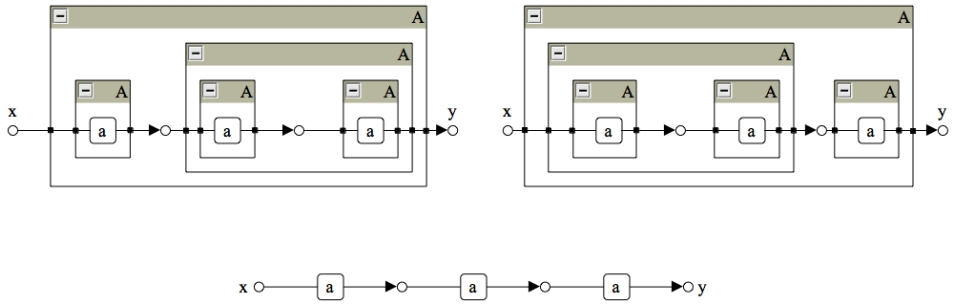

Figure 3: Sequence composition without (top) and with flattening (bottom).

and $X;Y \stackrel{\text{def}}{=} A_{u,v}[(\nu w)(X\langle u,w\rangle \mid Y\langle w,v\rangle)]$, where X and Y have type A. The graphical representation of both operators is visualised in Figure 2. We put the operator declaration on the top bar of the outermost design and we annotate the variables with their names and types. Note that, implicitly, the type of the outermost box is the type returned by the operation. Clearly, the algebra as such constructs hierarchical sequences, where e.g. $(\alpha;(\alpha;\alpha))\langle x,y\rangle$ and $((\alpha;\alpha);\alpha)\langle x,y\rangle$ intuitively define different graphs due to the nestings (see Figure 3). If we introduce the flattening axiom $\mathsf{flat}_{\mathsf{A}}$ in the algebra, instead, the two former terms are identified, and intuitively correspond to the normal form $(\nu w_1,w_2)(a(x,w_1) \mid a(w_1,w_2) \mid a(w_2,y))$ (see Figure 3).

The above example illustrates the two roles of the nesting operator: as a way 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. The example also illustrates how graphical encodings of existing (algebraic) languages are defined and exploited: the main issue is to see the constructors of the original language as derived operators of the graph algebra. Note that this enables the use of term rewrite techniques at the level of the original language. Consider for instance the term rewrite rule $X;Y \Rightarrow Y;X$ for the above example, where $X,Y:A$. With just one rule we are capturing all the possible ways to permute two arbitrary connected subsequences (the rule is applicable in any larger term and under any substitution of X and Y by terms of type A). Or else, consider proving by structural induction that the obtained graphs are all connected sequences. Such simplicity cannot be achieved easily with ordinary set-theoretic presentations of graphs.

Another kind of axioms that may be useful to include in the structural congruence are *extrusion* axioms. It is worth to mention that the extrusion axiom was not presented in [4] since it was not needed for the examples there, while in [5] extrusion for all design labels was considered as part of the structural congruence as it was used in all the examples. To the contrary we see here examples where extrusion is needed for some labels only.

Definition 7 (extrusion axiom) *An extrusion axiom* $\mathsf{extr}_{\mathsf{L}}$ *for some design label* L *is of the form* $L[(\nu y)\mathbb{G}]\langle\overline{x}\rangle \equiv (\nu y)L[\mathbb{G}]\langle\overline{x}\rangle$*, where* $y \notin \lfloor\overline{x}\rfloor$*.*

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 happens for some process calculi. Indeed, we see in § 7 how these 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 either exactly one or none of the axioms $\mathsf{flat}_{\mathsf{L}}$ and $\mathsf{extr}_{\mathsf{L}}$ is present.

3 A model of hierarchical graphs

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 alphabets $\mathcal{E}$ and $\mathcal{D}$.

Definition 8 (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 9 (hierarchical graph) *The set* $\mathcal{H}$ *of* hierarchical graphs *is the smallest set*[3] *containing the tuples* $G = \langle N_G, E_G, t_G, i_G, x_G, r_G, F_G\rangle$ *where*

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

1. $\langle N_G, E_G, t_G, F_G \rangle$ *is a graph layer;*

2. $i_G : E_G \cap \mathcal{A}_\mathcal{D} \to \mathcal{H}$ *is an embedding function (we say that* $i_G(e)$ *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) $\forall n, m \in \mathbb{N}$ *we have that* $x_G(e)[n] = x_G(e)[m]$ *iff* $t_G(e)[n] = t_G(e)[m]$*, i.e. it is not possible to expose a node twice without attaching it to the same external node (and vice versa);*

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 [14], with i_G 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 [18]. In practice, we shall often assume that $r_G(e)$ (when defined) is the ordinary inclusion: the general case is useful to embedd and reuse graphs without renaming their nodes.

Recalling the programming metaphor, intuitively each hierarchical edge e can be seen as a procedure declaration: $t_G(e)$ are the actual arguments, $x_G(e)$ the formal parameters, $F_{i_G(e)}$ the global variables for which $r_G(e)$ acts as aliasing, and $N_{i_G(e)} \setminus (F_{i_G(e)} \cup \lfloor x_G(e) \rfloor)$ the local variables.

Example 3. Consider the last term of Example 1 and its informal graphical representation on Figure 1 (right). Its actual interpretation as a hierarchical

[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.

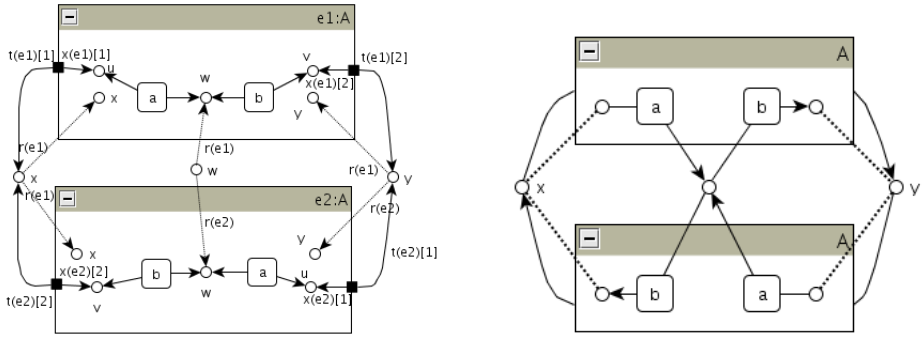

Figure 4: A hierarchical graph (left) and its simplified representation (right)

graph appears in Figure 4 (left) decorated with the most relevant annotations (the tentacle, exposition and renaming functions for the two hierarchical edges). As witnessed by Figure 4 (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 (reminiscent of Figure 2 (right)) that still retains all the relevant information. Note that such a simplified representation is very close to the informal notation of terms of our graph algebra shown in Figure 1.

These examples give an intuition about how our model of hierarchical graphs works and the comparison with the informal representation suggest how they could be used to obtain an intuitive, clear visualisation. The examples should also highlight that the algebra defined in § 2 is providing a simple syntax that hides the complexities of hierarchical structures, as it occurs in our model of hierarchical graphs. 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 actual graphs.

In the rest of the section we explain how such graphs are obtained out of terms, but first we have to fix some notation and concepts. In the following, we shall just use *graph* in place of *hierarchical graph*. Note that

the embedding structure forms a directed acyclic graph, whose unfolding we call *embedding tree*. The leaves of the embedding tree are actually plain graphs. The *height* (resp. *depth* or *layer*) of a graph is the height (resp. depth) of its embedding tree. In the following, $\mathcal{H}$ denotes both the set of all such graphs or the category having such graphs as objects and the following graph morphisms as arrows.

Definition 10 (graph morphism) *Let G, H be graphs such that $F_G \subseteq F_H$. A* graph morphism *$\phi : G \to H$ is a tuple $\langle \phi_N, \phi_E, \phi_I \rangle$ where $\phi_N : N_G \to N_H$ is a node morphism, $\phi_E : E_G \to E_H$ an edge morphism, and $\phi_I = \{\phi^e \mid e \in E_G \cap \mathcal{A}_\mathcal{D}\}$ a family of graph morphisms $\phi^e : i_G(e) \to i_H(\phi_E(e))$ such that*[5]

1. *$\forall e \in E_G$, $\phi_N(t_G(e)) = t_H(\phi_E(e))$, i.e. the tentacle function is respected;*
2. *$\forall e \in E_G \cap \mathcal{A}_\mathcal{D}$, $\phi^e_N(x_G(e)) = x_H(\phi_E(e))$, i.e. the exposure function is respected;*
3. *$\forall e \in E_G \cap \mathcal{A}_\mathcal{D}$, $\forall n \in N_G$, $\phi^e_N(r_G(e)(n)) = r_H(\phi_E(e))(\phi_N(n))$, i.e. the renaming function is respected;*
4. *$\forall n \in F_G$, $\phi_N(n) = n$, i.e. the free nodes are preserved.*

In the above definition we abuse the notation by lifting morphisms to sets and vectors. It is worth to observe that our morphisms are not the most general one can define. In particular, using the terminology of [24] they are *root-level* in the sense that they represent a layer-by-layer embedding. More general notions are the *deep* morphisms of [24] which embed a graph G into some lower graph of the embedding tree of a graph H. However, for the purpose of this paper our morphisms are enough: we can easily define isomorphisms and the category obtained has all pushouts for spans of injective morphisms, which we use to define a composition operator and which prepare the ground for some basic pushout-based graph transformations.

Proposition 2 (pushouts [13]) *Let $\phi : G \to H$, $\psi : G \to I$ be injective graph morphisms. Then, the pushout of ϕ and ψ always exists.*

Here, injectiveness simply means that the underlying functions on the nodes and edges of the graph layers are also injective. The proof is then easy, since no item coalescing is forced by the span of arrows, and all the auxiliary functions (exposure, etc.) are defined in the expected way.

[5] Again, many-sorted alphabets would require the morphisms to be type consistent.

4 Encoding into graphs.

We describe here the algebraic characterisation of graphs. We start presenting a few auxiliary definitions. In the following we denote the empty function with $\bot$, distinguishing it from the empty set $\emptyset$.

Definition 11 *Let $N \in \mathcal{N}$ be a subset of the nodes of graph G. Then, $\widehat{N}$ is the hierarchical graph given by the tuple $\langle N, \emptyset, \bot, \bot, \bot, \bot, N\rangle$, and $in_N : \widehat{N} \to G$ is the obviously defined, injective graph morphism.*

Graph composition is always defined, thanks to Proposition 2.

Definition 12 (graph composition) *Let G, H be graphs. Then, the* composition *of G and H, denoted $G \oplus H$, is the (codomain of the) pushout of the span $\widehat{F_G \cap F_H} \to G$ and $\widehat{F_G \cap F_H} \to H$.*

We are now ready to see how terms of our algebra can be interpreted as graphs. 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 13 (graph interpretation) *The encoding $[\![\cdot]\!]$, mapping well-formed terms into graphs, is the function inductively defined as*

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

where $e' \in \mathcal{A}_\mathcal{E}$ and $e \in \mathcal{A}_\mathcal{D}$.

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 row) 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.

We say that two graphs G,H are isomorphic (denoted $G \simeq H$) whenever there is an isomorphism between them.

We can now show that our encoding is sound and complete, meaning that equivalent terms are mapped to isomorphic graphs and vice versa.

Theorem 1 *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 $[\![\mathbb{G}_1]\!] \simeq [\![\mathbb{G}_2]\!]$.*

Proof: The soundness is rather straightforward: it proceeds by showing that each axiom is preserved by the encoding. The result follows from standard properties of pushouts for axioms DA1–DA3 and axiom DA9 and of set difference for axioms DA4–DA6. Alpha-renaming axioms DA7–DA8 are dealt with by graph isomorphism, thanks to the side conditions in Definition 4 that guarantee type preservation and avoid node coalescing. The most interesting axiom is therefore DA10, for which we detail the proof. Below we let $[\![\mathbb{G}]\!] = \langle N_{\mathbb{G}}, E_{\mathbb{G}}, t_{\mathbb{G}}, i_{\mathbb{G}}, x_{\mathbb{G}}, F_{\mathbb{G}}\rangle$, $N' = (F_{\mathbb{G}} \setminus \lfloor \overline{x} \rfloor) \cup \lfloor \overline{y} \rfloor$ and $N'' = N' \cup \{z\}$.

$$
\begin{array}{ll}
& [\![z \mid L_{\overline{x}}[\mathbb{G}]\langle \overline{y}\rangle]\!] \\
= & [\![z]\!] \oplus [\![L_{\overline{x}}[\mathbb{G}]\langle \overline{y}\rangle]\!] \\
= & [\![z]\!] \oplus \langle N', \{e\}, e \mapsto \overline{y}, e \mapsto [\![\mathbb{G}]\!] \oplus [\![\lfloor \overline{y} \rfloor]\!], e \mapsto \overline{x}, e \mapsto id_{N'}, N'\rangle \\
\simeq & \langle N'', \{e\}, e \mapsto \overline{y}, e \mapsto [\![z]\!] \oplus [\![\mathbb{G}]\!] \oplus [\![\lfloor \overline{y} \rfloor]\!], e \mapsto \overline{x}, e \mapsto id_{N''}, N''\rangle \\
= & \langle N'', \{e\}, e \mapsto \overline{y}, e \mapsto [\![z \mid \mathbb{G}]\!] \oplus [\![\lfloor \overline{y} \rfloor]\!], e \mapsto \overline{x}, e \mapsto id_{N''}, N''\rangle \\
= & [\![L_{\overline{x}}[z \mid \mathbb{G}]\langle \overline{y}\rangle]\!]
\end{array}
$$

The crucial step is the one where graph isomorphism $\simeq$ is exploited: the passage is valid, because z is a free name and therefore it is certainly preserved by the morphisms induced from the pushout at the top-level.

The completeness proof is more involved. The proof sketch we present here is modelled after the one for [11, Lemma 22], and it proceeds by exploiting the normal form for well-formed terms of our algebra (see Definition 5).

Now, let $\mathbb{G}_1$ and $\mathbb{G}_2$ be two terms such that $\mathbb{G}_1 \not\equiv_D \mathbb{G}_2$ but they are mapped to isomorphic graphs. Without loss of generality, we assume that the terms are in normal form and that the sum of their depths is the minimal value for which two such terms can be found. The isomorphism $\phi : [\![\mathbb{G}_1]\!] \to [\![\mathbb{G}_2]\!]$ ensures that at the top level the graphs $[\![\mathbb{G}_1]\!]$ and $[\![\mathbb{G}_2]\!]$ have the same number of nodes and exactly the same free nodes. Moreover it establishes a bijective correspondence between the edges in $[\![\mathbb{G}_1]\!]$ and those in $[\![\mathbb{G}_2]\!]$, so that e and $\phi(e)$ must carry the same label and their tentacle functions must commute w.r.t. ϕ. Thus, $\mathbb{G}_1$ and $\mathbb{G}_2$ must have the shape

$$\mathbb{G}_1 = (\nu x_1) \ldots (\nu x_m) \left(z_1 \mid \ldots \mid z_n \mid \Pi_j l_j \langle \overline{y}_j \rangle \mid \Pi_i L^i_{\overline{x}_i}[\mathbb{G}'_i]\langle \overline{w}_i \rangle\right)$$

and

$$\mathbb{G}_2 = (\nu x'_1)\dots(\nu x'_m)\left(z'_1 \mid \dots \mid z'_n \mid \Pi_j l_j\langle\overline{y'}_j\rangle \mid \Pi_i L^i_{\overline{x'}_i}[\mathbb{G}''_i]\langle\overline{w'}_i\rangle\right)$$

for suitable $\mathbb{G}'_i$ and $\mathbb{G}''_i$ in normal forms such that $\mathbb{G}'_i \simeq \mathbb{G}''_i$. Since $\mathbb{G}_1 \not\equiv_D \mathbb{G}_2$, there must be some index k such that $\mathbb{G}'_k \not\equiv_D \mathbb{G}''_k$, but this contradicts the existence of $\mathbb{G}_1$ and $\mathbb{G}_2$, because the sum of the depths of $\mathbb{G}'_k$ and $\mathbb{G}''_k$ is clearly strictly less than that of $\mathbb{G}_1$ and $\mathbb{G}_2$. □

Moreover, our encoding is surjective, i.e. every graph can be denoted by a term of the algebra.

Proposition 3 *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 $[\![\mathbb{G}]\!]$.*

Proof: The proof proceeds by induction on the height of the embedding tree of a graph.

If the height is 0, i.e. if the graph is flat, the proof is quite straightforward. Indeed, let us consider a graph $\langle N, E, \bot, \bot, \bot, F\rangle$. Now, the underlying graph without interfaces can be considered as the union of (possibly connected) edges $\{l_1\langle\overline{x}_1\rangle \dots l_k\langle\overline{x}_k\rangle\}$ and isolated nodes $\{y_1, \dots, y_m\}$, additionally verifying $F \subseteq N = \bigcup_{i=1\dots k}\lfloor\overline{x}_i\rfloor \cup \{y_1, \dots, y_m\}$. Thus, the associated term of the algebra is given by $(\nu z_1)\dots(\nu z_n)(N \mid \Pi_{i=1\dots k} l_i\langle\overline{x}_i\rangle)$ for $\{z_1 \dots z_l\} = N \setminus F$. Performing the parallel composition by means of pushouts implements the sharing of nodes among the edges of the graph.

The induction step is similar. Let us assume that the correspondence holds for each graph in the lower layers. Moreover, note that each one of those graphs contains as free nodes all those nodes occurring at the top-most layer. And since also the top-most layer of the graph can be modeled as the union of (possibly connected) edges and isolated nodes, the required term is obtained by inserting all the terms corresponding to the graphs in the lower layers in the right position of the design at the top level. □

If either flattening or extrusion axioms are present, then the encoding must be changed to "dissolve" certain embeddings and edges. To this aim, we need to distinguish three different cases in the encoding of designs: the first rule works exactly as before, when neither flattening nor extrusion axioms are actually present, while the other two are shown below

$$\begin{array}{lcll} [\![L_{\overline{x}}[\mathbb{G}]\langle\overline{y}\rangle]\!] & = & \langle N, \{e\}, e \mapsto \overline{y}, e \mapsto [\![\mathbb{G}]\!] \oplus \widehat{\lfloor\overline{y}\rfloor}, e \mapsto \overline{x}, e \mapsto id_N, N'\rangle & \text{if } \mathsf{extr}_{\mathsf{L}} \in \equiv_{\mathrm{D}} \\ [\![L_{\overline{x}}[\mathbb{G}]\langle\overline{y}\rangle]\!] & = & [\![\mathbb{G}\{^{\overline{y}}/_{\overline{x}}\}]\!] & \text{if } \mathsf{flat}_{\mathsf{L}} \in \equiv_{\mathrm{D}} \end{array}$$

where $e : \mathcal{A}_{\mathcal{D}}$ and N, N' stand for $(N_{\mathbb{G}} \setminus \lfloor \overline{x} \rfloor) \cup \lfloor \overline{y} \rfloor$ and $(F_{\mathbb{G}} \setminus \lfloor \overline{x} \rfloor) \cup \lfloor \overline{y} \rfloor$, respectively.

If a flattening axiom occurs, there is no associated edge: the encoding of $L_{\overline{x}}[\mathbb{G}]\langle \overline{y} \rangle$ is the same as the one of $\mathbb{G}$, after suitably renaming the nodes. In other words, the axiom is interpreted directionally, and the associated enclosing edge has no occurrence of the flattened design. Likewise, if an extrusion axiom occurs, the structural congruence is interpreted directionally, and the restriction operators float to the top. Indeed, now all the names of $\mathbb{G}$ appears in $L_{\overline{x}}[\mathbb{G}]\langle \overline{y} \rangle$, except those in $\lfloor \overline{x} \rfloor$ (exposed as those in $\lfloor \overline{y} \rfloor$).

Soundness and completeness still hold. However, in the presence of extruding axioms the encoding is not surjective, unless we impose some well-formedness criteria over embedding edges to require that all the nodes of a lower layer that are not exposed do occur in the higher one when the embedding exploits an arc with label L for which the extruding axiom holds.

5 A calculus with flat structure and communication: the π-calculus

This section offers a first instance of our approach by presenting an encoding of the finitary fragment of the π-calculus [22]. We have chosen this example due to its popularity and simplicity. Moreover, as a graphical encoding already exists [18], we can compare the two proposals and evaluate the convenience of exploiting our graph algebra in the definition of the encoding and, most importantly, on its proof of correctness. Familiarity with the calculus would be helpful but our presentation should suffice for our aims.

5.1 The π-calculus

Definition 14 (π-calculus syntax) *Let $\mathcal{U}$ be a set of* names. *The set $\mathcal{P}$ of (finite)* processes *is the set of terms of sort P generated by the grammar*

$$\begin{array}{lcl} P & ::= & \mathbf{0} \mid M \mid (\nu a)P \mid P \mid P \\ M & ::= & M + M \mid \pi.P \end{array}$$

where $\pi \in \{\tau\} \cup \{a(b), \overline{a}b \mid a, b \in \mathcal{U}\}$ and $a \in \mathcal{U}$.

In the definition above, terms generated by P and M are called *process* and *sequential processes* (or *summations*), respectively. We recall that τ, $a(b)$ and $\overline{a}b$ are called, respectively, the silent prefix, the input prefix and the

output prefix; moreover, the input prefix $a(b).P$ and the restriction operator $(\nu b)P$ act as binders for b with scope P. We denote by $n(\pi)$ the names appearing in π, i.e. $n(\tau) = \emptyset$ and $n(a(b)) = n(\overline{a}b) = \{a, b\}$. The standard definition for the set of free names of a process P, denoted by $fn(P)$, is assumed. Similarly for α-convertibility, with respect to the binders: the name b is bound in both $a(b).P$ and $(\nu b)P$, and it can be freely α-converted.

Example 4. Consider the following process **agent** $\stackrel{\text{def}}{=}$ (νsecret) $\overline{\text{private}}$ secret which represents an agent ready to send a fresh name *secret* over the free channel *private*. Now, consider **gossiper** $\stackrel{\text{def}}{=}$ (νmsg) private(confidential). ($\overline{\text{confidential}}$ msg + $\overline{\text{public}}$ confidential), which represents an agent ready to read a name *confidential* from the free channel *private*, after which a new message *msg* over the *confidential* channel or the *confidential* name over the free channel *public* can be sent. Note that $fn(\textbf{agent}) = \{\text{private}\}$, while $fn(\textbf{gossiper}) = \{\text{private}, \text{public}\}$. The process **sys** $\stackrel{\text{def}}{=}$ (νprivate)(**agent** | **gossiper**) represents a system in which the former two agents are put in parallel and communicate through the local channel *private* (in fact $fn(\textbf{sys}) = \{\text{public}\}$).

A congruence relation captures structural equivalences like the commutativity and associativity of parallel composition or the α-renaming of bound names. The structural congruence for the π-calculus is defined as follows.

Definition 15 (π-calculus structural congruence) *The* structural congruence *for processes* $\equiv_\pi$ *is the least congruence satisfying*

$$
\begin{array}{rcll}
P \mid Q & \equiv & Q \mid P & (\pi\mathsf{A1}) \\
P \mid (Q \mid R) & \equiv & (P \mid Q) \mid R & (\pi\mathsf{A2}) \\
P \mid \mathbf{0} & \equiv & P & (\pi\mathsf{A3}) \\
M + N & \equiv & N + M & (\pi\mathsf{A4}) \\
M + (N + O) & \equiv & (M + N) + O & (\pi\mathsf{A5}) \\
(\nu a)\mathbf{0} & \equiv & \mathbf{0} & (\pi\mathsf{A6}) \\
(\nu a)(\nu b)P & \equiv & (\nu b)(\nu a)P & (\pi\mathsf{A7}) \\
P \mid (\nu a)Q & \equiv & (\nu a)(P \mid Q) \quad \textit{if } a \notin fn(P) & (\pi\mathsf{A8}) \\
\pi.(\nu a)P + M & \equiv & (\nu a)(\pi.P + M) \quad \textit{if } a \notin fn(M) \cup n(\pi) & (\pi\mathsf{A9}) \\
(\nu a)P & \equiv & (\nu b)P\{^b/_a\} \quad \textit{if } b \notin fn(P) & (\pi\mathsf{A10}) \\
c(a).P & \equiv & c(b).P\{^b/_a\} \quad \textit{if } b \notin fn(P) & (\pi\mathsf{A11})
\end{array}
$$

Axiom πA9 is not standard but is sometimes included in the structural congruence in order to consider restriction as some sort of declaration (conversely, the absence of the axiom means that restriction is some sort of run-time fresh name generation). In our case we preferred to consider it

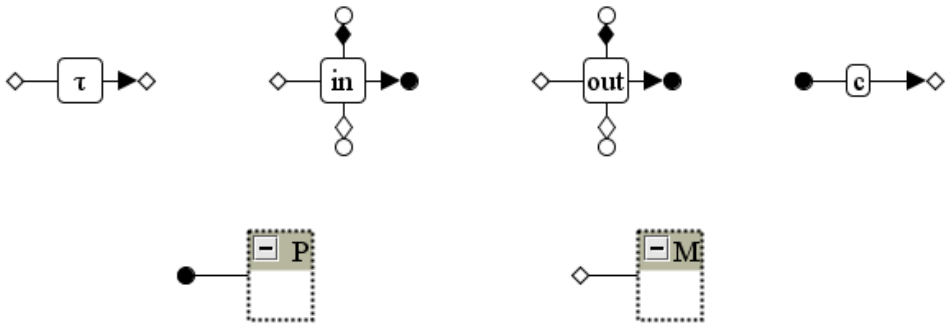

Figure 5: Type graph for the π-calculus.

since it yields a more clear graphical representation. However, dealing with the absence of the axiom is also standard as explained in [18].

5.2 Encoding the π-calculus

We start presenting (cf. Figure 5) the graph items that we shall use. Basically, we have design sorts (labels of $\mathcal{D}$) corresponding to those present in the π-calculus, i.e. the syntactical categories for processes (P) and summations (M) and the sort of 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 parallel and sequential processes, and channel names, respectively. Design labels P and M model designs representing parallel and sequential processes and they are used to ensure the well-formedness of graphs. To achieve this we introduce the flattening axioms $\mathsf{flat_P}$ and $\mathsf{flat_M}$, which in the visualisation is represented by using dotted boxes. Both design types P and M have a unique tentacle denoted with a plain line, which is to be attached to a control point of the corresponding type.

Edge labels of ($\mathcal{E}$) are τ, in, out and c that respectively correspond to silent actions, inputs, outputs and explicit casting from sequential to parallel processes (to be explained later). Such labels are needed because we consider action prefixes as being *material* in the encoding, i.e. we use graph items to represent them. On the other hand, parallel composition and non-deterministic choice are considered as being *immaterial*, i.e. they are interpreted as graph operations that do not introduce any graph item. Intuitively, this reflects the axioms associated to the operations. We use a plain line and an arrow for the entry and exit control points of actions and the

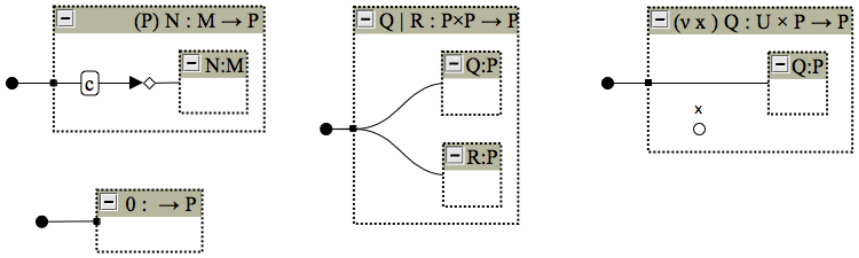

Figure 6: Graphical interpretation for the π-calculus (processes).

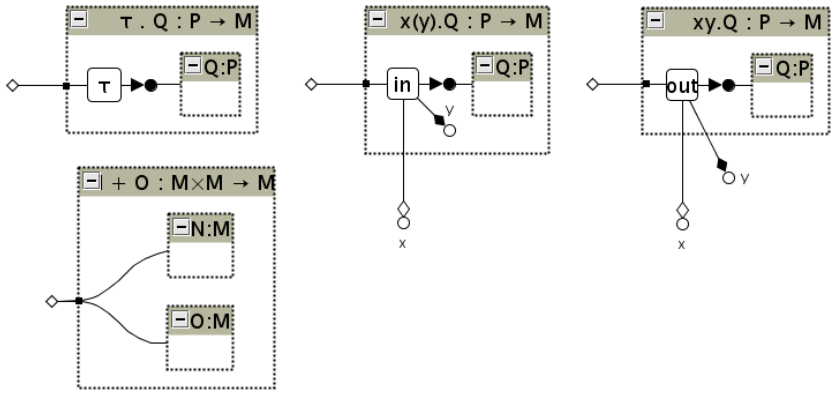

Figure 7: Graphical interpretation for the π-calculus (summations).

explicit cast. Channel and message arguments of communication operations are denoted by arrows ended by empty and filled diamonds, respectively.

We are ready to define the graphical encoding of the π-calculus. We define it in terms of derived operators, instead of using a denotational encoding, to stress the similarities and common sorting between the calculus and our graph algebra. We find convenient to introduce a cast operator from M to P (as in [18]) which allows to distinguish between the two sorts of control points where different forms of branching apply (parallel and choice).

Definition 16 (π-calculus encoding) *The interpretation of the operators*

of the π-calculus over the design algebra is given by

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

together with axioms flat_P *and* flat_M.

The graphical representation of the above definition can be found in Figures 6 and 7. We remark only the most relevant aspects. Casting from sequential processes into parallel ones must be done explicitly to distinguish summations from processes (as in [18]). This is done by connecting via prefixing the graph of a sequential process with a c-labelled edge. The parallel composition of two processes Q and R amounts to embed the respective graphs of Q and R in P-typed edges attached to the same $\bullet$-typed node. Note that our informal visualisation fuses the tentacles of the designs corresponding to the two processes that are put in parallel and the resulting one (their composition). This results in a visually appealing representation. The presence of the flattening axiom flat_P will dissolve the embedding and as a result the corresponding graphs will be at the same level. Processes in parallel thus become graphs departing from the same control point. Another relevant part of the encoding regards the input prefix, since it involves a free and a bound name, and a process. We see that the encoding of $x(\overline{y}).Q$ consists of an arc representing the input operation which is attached to the main $\diamond$-typed control point and its $\bullet$-typed continuation where the graph corresponding to Q is plugged. The edge representing the input action is connected to a free and a bound node representing the communication channel x and the argument channel y, respectively. In our visualisation, variable graph items are denoted by labelling them with variable names (such as x and y in the encoding of action prefixes).

Example 5. Recall the process of Example 4. Its graphical encoding is depicted on Figure 8. The figure clearly represents the two different forms of branching: the parallel composition of both the *agent* and the *gossiper* processes and the choice of the *gossiper* after reading the *secret* channel.

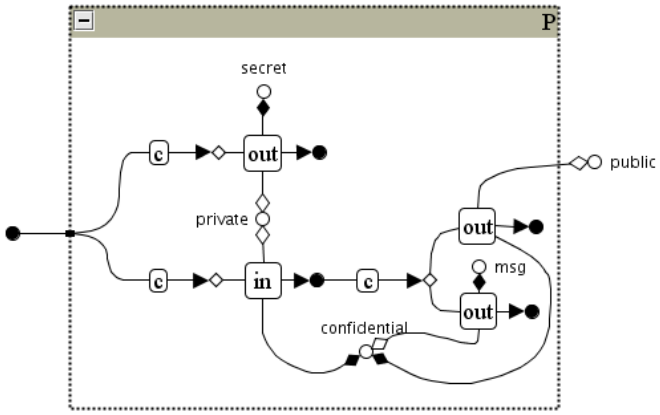

Figure 8: Graphical encoding of a process.

Note how the sort of edges disambiguates the form of branching and how explicit casting is used to change the control point sort. The sharing of names (like channel *private*) where processes synchronise is also evident. It is also worth noting how the graphical representation distinguishes global and restricted names: the former are depicted as lying outside the P-labeled frame and the latter inside it.

The proposed encoding is sound and complete, i.e. two processes are structurally congruent if and only if they are mapped to isomorphic graphs. As a matter of fact, the graphs obtained are roughly the same as those proposed in the encoding of [18] for the finite fragment of the calculus. In addition, our encoding precisely characterises in a compact and elegant way the set of all graphs that correspond to π-calculus processes, namely those generated by the derived algebra (which is implicitly given by the encoding).

Proposition 4 (correctness of π-calculus encoding) *For any* $P, Q \in \mathcal{P}$, $P \equiv_\pi Q$ *iff* $P \equiv_{\mathrm{D}} Q$.

Proof: The graph algebra provides a handy, elegant notation to carry out the proof of soundness in a purely algebraic form. For the purpose of the proof it turns out to be convenient to use a functional notation for the encoding. So we let $[\![P]\!]$ denote the interpretation of P according to

Definition 16. Now, all we have to show is that the structural axioms of the π-calculus identify equivalent designs. More precisely, we have to show that for each axiom $P \equiv Q$ in $\equiv_\pi$ we have $[\![P]\!] \equiv_{\mathrm{D}} [\![Q]\!]$. This is enough since the proof of $P \equiv Q$ just applies the axioms of $\equiv_\pi$, plus additionally the closure with respect to the operators of the calculus, and the latter component is satisfied by definition.

Consider axiom $\pi\mathsf{A1}$, i.e. $Q \mid R \equiv R \mid Q$. We have

$$\begin{array}{lll} & [\![Q \mid R]\!] & \\ \equiv & P_p[Q(p)|R(p)] & \text{(Definition 16)} \\ \equiv & P_p[R(p)|Q(p)] & \text{(Axiom } \mathsf{DA1}\text{)} \\ \equiv & [\![R \mid Q]\!] & \text{(Definition 16)} \end{array}$$

The proof for the remaining axioms is similar. For instance, the proof for axioms $\pi\mathsf{A2}-\pi A6$ regarding commutativity, associativity and neutral element for parallel and non-deterministic composition of processes is straightforward (as the above one of $\pi\mathsf{A1}$) as we have similar axioms for parallel composition of graphs in $\equiv_{\mathrm{D}}$. The same reasoning can be applied for α-conversion and for axioms $\pi\mathsf{A7} - \pi\mathsf{A9}$ regarding the restriction operator.

As the encoding maps processes to essentially flat graphs, the proof of completeness could be carried out just exploiting the result in [18]. However, we provide here a direct proof that exploits algebraic reasoning: we shall use the normal form of π-calculus processes to show that $P \not\equiv_\pi Q \Rightarrow [\![P]\!] \not\equiv_{\mathrm{D}} [\![Q]\!]$. The standard normal form of processes is $(\nu\overline{x})S_1 \mid \cdots \mid S_n$, where $\overline{x} \subseteq fn(S_1 \mid \cdots \mid S_n)$ and each S_i is of the form $\sum_{j=1}^{n_i} A_{i,j}.Q_{i,j}$ with each $Q_{i,j}$ again in normal form, but with no occurrence of the restriction operator: intuitively, all restrictions appear as early as possible in the term and what follows is the parallel composition of non-deterministic choices of processes $Q_{i,j}$ in normal form, all prefixed with an action $A_{i,j}$.

Now suppose that we are given two processes Q and R that are not structurally equivalent, i.e. we have $Q \not\equiv_\pi R$. We analyse all possibilities for this to occur and show that in all cases it follows $[\![Q]\!] \not\equiv_{\mathrm{D}} [\![R]\!]$. Roughly, either the two processes have the same outermost shape or they do not. If they have the same outermost shape, then they must differ for some subterms and then we can exploit inductive hypothesis to assume that such subterms correspond to non isomorphic graphs and then conclude that $[\![Q]\!] \not\equiv_{\mathrm{D}} [\![R]\!]$ (the base case for induction is vacuous, as both processes would be $\mathbf{0}$). Therefore we are left to show that if Q and R have different outermost shapes their encodings can be distinguished.

We start with the simple case where the topmost restriction differ. Without loss of generality suppose that $Q \stackrel{\text{def}}{=} (\nu\overline{x})Q'$ and $R \stackrel{\text{def}}{=} (\nu x)(\nu\overline{x})R'$, with x in $fn(R')$ but not in $\overline{x}$. We have $[\![Q]\!] = P_p[(\nu\overline{x})[\![Q']\!]\langle p\rangle]$ and $[\![R]\!] = P_p[(\nu x)(\nu\overline{x})[\![R']\!]\langle p\rangle]$: they cannot be identified by $\equiv_{\mathrm{D}}$ since x is clearly part of $fn([\![R']\!])$ and we know that $\equiv_{\mathrm{D}}$ respects free names.

Consider now that both top-restrictions are equivalent but the number of sequential processes in the topmost parallel differs. Without loss of generality suppose that $Q \stackrel{\text{def}}{=} (\nu\overline{x})(S_1 \mid \cdots \mid S_n)$ and $R \stackrel{\text{def}}{=} (\nu\overline{x})(T_1 \mid \cdots \mid T_n \mid T_{n+1})$. We have $[\![Q]\!] = P_p[(\nu\overline{x})([\![S_1]\!]\langle p\rangle \mid \cdots \mid [\![S_n]\!]\langle p\rangle)]$ and $[\![R]\!] = P_p[(\nu\overline{x})([\![T_1]\!]\langle p\rangle \mid \cdots \mid [\![T_n]\!]\langle p\rangle \mid [\![T_{n+1}]\!]\langle p\rangle)]$: they cannot be equivalent terms since each $[\![S_i]\!]$ (being of the form $\sum_{j=1}^{n_i} A_{i,j}.Q_{i,j}$) contributes with at least one distinguished tentacle outgoing from node p, and similarly for each $[\![T_i]\!]$.

The rest of the cases follow a similar reasoning. □

6 A calculus with nested structure and no communication: Sagas

We consider in this section the *nested sagas with programmable compensations* of [8], a calculus for long running transactions.

6.1 Sagas

The calculus (which we shall call just *sagas*) 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 17 (sagas syntax) *Let $\mathcal{A}$ be a set of atomic activities. The sets $\mathcal{S}$ of sagas and $\mathcal{P}$ of compensable processes are the sets of terms of sorts S and P, respectively, generated by the grammar below*

$$\begin{array}{lcll} S & ::= & a \mid \{P\} & \textit{(sagas)} \\ P & ::= & S\%S \mid P;P \mid P \mid P & \textit{(processes)} \end{array}$$

where $a \in \mathcal{A}$.

For the sake of simplicity, with respect to the original presentation we neglect the introduction of nil processes and non-compensable activities. A *saga* is an atomic activity or an arbitrarily complex transaction built out

from a compensable process. 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).

Example 6. Consider the following example, inspired from [8] 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. 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 absence of a reward account (idem for activity *substractPoints*).

The structural congruence for sagas captures the associativity of sequential and parallel composition and the commutativity of the latter.

Definition 18 (sagas structural congruence) *The* structural congruence *for sagas* $\equiv_{\mathrm{S}}$ *is the least congruence satisfying*

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

where $P, Q, R \in \mathcal{P}$

6.2 Encoding sagas

As in the encoding of the π-calculus the first idea is to interpret syntactical categories of the calculus as design sorts (i.e. labels in $\mathcal{D}$) 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 is a subsort of S, while P is a subsort for T. Figure 9 illustrates our type graph. We have

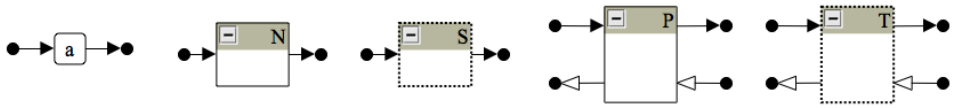

Figure 9: Type graph for sagas

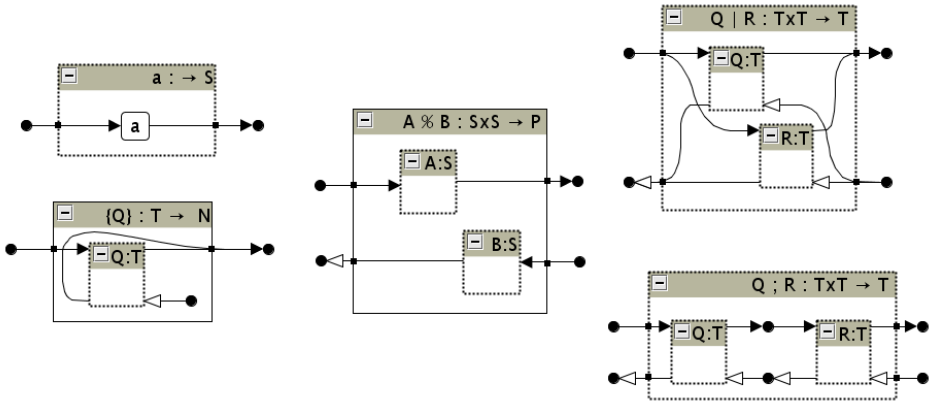

Figure 10: Graphical interpretation for sagas.

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 $\mathcal{A}$, i.e. $\mathcal{A}$ is our designed set of edge labels $\mathcal{E}$. Because S and T are just used for composition we introduce the flattening axioms flat_S and flat_T.

The encoding is formally defined as follows (c.f. Figure 10).

Definition 19 (sagas encoding) *The interpretation of the sagas opera-*

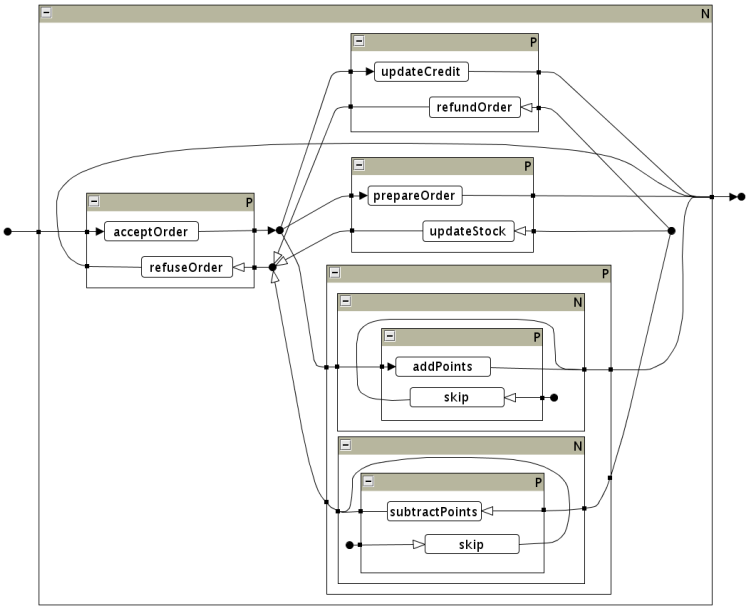

Figure 11: Graphical encoding of a saga

tors over the design algebra is given by

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

together with axiom $\mathsf{flat}_{\mathsf{S}}$ *and* $\mathsf{flat}_{\mathsf{T}}$.

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 it was the case for actions in the π-calculus) as shown in Figure 9 and also of compensable pairs and nested sagas. Instead, sequencing and parallel composition (see Figure 10) are immaterial and their associated axioms are captured by the flattening axioms.

Example 7. Figure 11 depicts the graphical representation of the saga introduced in Example 6. It is worth to note the nesting of sagas which

decouples the entry of the compensation flow and redirects the exit flow into the ordinary flow. Also note the two uses of nesting: immaterial for parallel and sequential composition and material for basic processes and nested transactions.

The proposed encoding is sound and complete, i.e. equivalent processes and sagas are mapped into isomorphic graphs.

Proposition 5 (correctness of sagas encoding) *For any $Q, R \in \mathcal{P}$ we have $Q \equiv_{\mathrm{S}} R$ iff $Q \equiv_{\mathrm{D}} R$.*

Proof: As in the proof of the encoding of the π-calculus we show each direction of the equivalence separately, starting with soundness, i.e. $Q \equiv_{\mathrm{S}} R \Rightarrow Q \equiv_{\mathrm{D}} R$. Again, $[\![\cdot]\!]$ denotes the interpretation according to Definition 19 and we just need to show that the structural axioms of the sagas do only identify equivalent designs.

Consider axiom sA1, i.e. $P;(Q;R) \equiv (P;Q);R$. We have

$$
\begin{array}{ll}
 & P;(Q;R) \\
\equiv & T_{p,q,r,s}[(\nu u,v)(P\langle p,u,v,s\rangle \mid (Q;R)\langle u,q,r,v\rangle)] \\
 & \text{(Definition 19)} \\
\equiv & T_{p,q,r,s}[(\nu u,v)(P\langle p,u,v,s\rangle \mid T_{\langle u,q,r,v\rangle}[(\nu u',v')(Q\langle u,u',v',v\rangle \mid R\langle u',q,r,v'\rangle)])] \\
 & \text{(Definition 19)} \\
\equiv & T_{p,q,r,s}[(\nu u,v)(P\langle p,u,v,s\rangle \mid (\nu u',v')(Q\langle u,u',v',v\rangle \mid R\langle u',q,r,v'\rangle))] \\
 & (\mathsf{flat_T}) \\
\equiv & T_{p,q,r,s}[(\nu u',v')(\nu u,v)((P\langle p,u,v,s\rangle \mid Q\langle u,u',v',v\rangle) \mid R\langle u',q,r,v'\rangle)] \\
 & (\mathsf{DA4},\mathsf{DA6},\mathsf{DA2}) \\
\equiv & T_{p,q,r,s}[(\nu u',v')T_{\langle p,u',v',s\rangle}[(\nu u,v)(P\langle p,u,v,s\rangle \mid Q\langle u,u',v',v\rangle)] \mid R\langle u',q,r,v'\rangle] \\
 & (\mathsf{flat_T}) \\
\equiv & T_{p,q,r,s}[(\nu u',v')((Q;R)\langle p,u',v',s\rangle \mid R\langle u',q,r,v'\rangle)] \\
 & \text{(Definition 19)} \\
\equiv & (P;Q);R \\
 & \text{(Definition 19)}
\end{array}
$$

The proofs for axioms sA2 and sA3 are also straightforward and similar to the above proof and those for the parallel axiom of the π-calculus.

Now we prove completeness, i.e. $Q \equiv_{\mathrm{S}} R \Leftarrow Q \equiv_{\mathrm{D}} R$. The proof technique is analogous to the one seen for π-calculus, but with a slightly more complicated case to consider, which requires an original bit of reasoning and where the graph algebra can be exploited conveniently. We shall use the normal form of processes to show that $Q \not\equiv_{\mathrm{S}} R \Rightarrow Q \not\equiv_{\mathrm{D}} R$. The normal form of a saga S is either a or $\{P\}$ with P in normal form and the normal form of a compensable processes Q is either $A\%B$ or $Q_1;\ldots;Q_n$ (with $n > 1$ and each Q_i again in normal form, of course excluding the occurrence of

sequence operators on top) or $Q_1 \mid \cdots \mid Q_n$ (with $n > 1$ and each Q_i again in normal form, excluding the occurrence of parallel compositions on top).

Now suppose that we are given two processes Q and R that are not structurally equivalent, i.e. we have $Q \not\equiv_{\mathrm{S}} R$. We analyse all possibilities for this to occur and show that in all the cases it follows $[\![Q]\!] \not\equiv_{\mathrm{D}} [\![R]\!]$. If they have the same shape, then the proof can be easily carried out by inductive arguments. If they have different shapes, then we must analyse the possible combinations separately.

We start with the simplest case for Q, i.e. $Q \equiv A\%B$. We have several cases in which R is not structurally equivalent to $A\%B$. First, R can be of the form $C\%D$ with at least one of C and D being respectively different from A and B. Trivially $A\%B \not\equiv_{\mathrm{D}} C\%D$. Another possibility for R is to be of the form $R_1; \ldots ; R_n$. This case is trivial since the interpretation of $\_;\_$ introduces new nodes that cannot be removed by $\equiv_{\mathrm{D}}$. Finally, R can be of the form $R_1 \mid \cdots \mid R_n$ but note that each subprocess R_i must have one of the forms we previously attempted for R (excluding the occurrence of parallel compositions on top). Hence, again they cannot be equivalent to $A\%B$. We conclude that $Q \not\equiv_{\mathrm{D}} R$.

The difficult case is when one of the processes (say Q) has the shape of a sequential composition $Q_1; \ldots ; Q_n$ (with $n > 1$) and the other process that of a parallel composition $R_1 \mid \cdots \mid R_m$ (with $m > 1$). In this case we can look more closely at the encoded terms. By structural induction it is easy to prove that both can be reduced to normal forms $T_{p,q,r,s}[(\nu U, V)\Pi_i P_{\overline{p}_i}[\mathbb{G}_i]\langle \overline{u}_i \rangle]$ and $T_{p,q,r,s}[(\nu U', V')\Pi_i P_{\overline{p}'_i}[\mathbb{G}'_i]\langle \overline{u}'_i \rangle]$. If they involve a different number of P-edges or a different number of restrictions then we are done. If not, let us observe that the parallel and sequential flows underlying the graphical representation of such processes induce a partial order over the nodes and edges (e.g. along the direction of the ordinary flow). Obviously such order must be preserved by graph isomorphism. Then we can prove that $Q_1; \ldots ; Q_n$ cannot be isomorphic to $R_1 \mid \cdots \mid R_m$ just by considering the fresh node u introduced by $[\![Q_1; (Q_2; \ldots ; Q_n)]\!] = T_{p,q,r,s}[(\nu u, v)([\![Q_1]\!]\langle p, u, v, s\rangle \mid [\![Q_2; \ldots ; Q_n]\!]\langle u, q, r, v\rangle)]$: clearly the node u must follow any (topmost) P-labeled edge introduced by Q_1, i.e. any (topmost) P-labeled edge with a tentacle attached to the node p. Now take $[\![R_1 \mid (R_2 \mid \cdots \mid R_m)]\!] = T_{p,q,r,s}[[\![R_1]\!]\langle p, q, r, s\rangle \mid [\![R_2 \mid \cdots \mid R_m]\!]\langle p, q, r, s\rangle]$. For $[\![Q_1; (Q_2; \ldots ; Q_n)]\!]$ and $[\![R_1 \mid (R_2 \mid \cdots \mid R_m)]\!]$ to be isomorphic, the (image of) node u should be introduced in $[\![R_1 \mid (R_2 \mid \cdots \mid R_m)]\!]$ either by $[\![R_1]\!]$ or by $[\![R_2 \mid \cdots \mid R_m]\!]$, but it is then evident that in any case there

would be P-labeled edges (at least one) provided by the other term which are attached directly to node p but are independent w.r.t. u (at the algebraic level, this is made clear by the scoping rules for restricted names). $\square$

7 A calculus with nested structures and communication: CaSPiS

This section presents the graphical representation of CaSPiS [1], a session-centered calculus. 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.

7.1 CaSPiS

We briefly overview CaSPiS and we refer the interested readers to [1] for an exhaustive description. We remark that we focus here on the close-free fragment of the calculus and we present a slightly different syntax. Both decisions are for the sake of a convenient and clean presentation and constitute no limitation.

Definition 20 (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 is the set of terms of sort P generated by the grammar*

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

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 all the above, each CaSPiS process can be thought of as running inside 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 8. Consider the process: $(\nu a)(\nu b)(a \triangleright (P_1 | b \triangleright P_2) | a \triangleright P_3 | b \triangleright P_4)$. This situation is typical: two sessions a and b have been created (as the result of two service invocations). Agent $a \triangleright (P_1 | b \triangleright P_2)$ participates to sessions a and b (assume P_1 is the protocol for a and P_2 the one for b), with the b side nested in a. The counter-party protocols for a and b are P_3 and P_4, respectively, and they run separately. Notably, values returned one level up by P_2 can be consumed by P_3.

Example 9. As another illustrative, typical example consider processes $P_1 > (P_2 > P_3)$, where each time P_1 emits a value an instance of $(P_2 > P_3)$ is generated (with P_3 being inactive). In any such instance, again, each value emitted by P_2 yields a new instance of P_3.

Next, we present the structural congruence for CaSPiS processes.

Definition 21 (CaSPiS structural congruence) *The structural congru-*

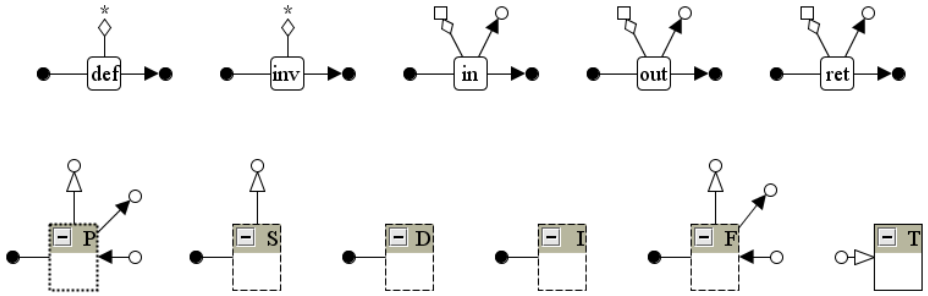

Figure 12: Type graph for CaSPiS.

ence for CaSPiS processes $\equiv_{\mathrm{C}} \subseteq \mathcal{P} \times \mathcal{P}$ *is the least congruence satisfying*

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

7.2 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 types can be found in Figure 12. 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 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 service callers and callees).

Definition 22 (CaSPiS encoding) *The interpretation of CaSPiS operators over the design algebra is given by*

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

together with axioms $\mathsf{flat_P}$, $\mathsf{extr_S}$, $\mathsf{extr_D}$, $\mathsf{extr_I}$ *and* $\mathsf{extr_F}$.

Part of the above definition is graphically represented in Figure 13. As in our previous examples we use different arrow types to denote the different (ordered, typed) tentacles of each edge. For example, for a design representing a process, an outgoing empty arrow represents its returning channel, an outgoing filled arrow its output channel, an incoming arrow its input channel and a plain arrow its control point. Again, arguments of an operation are denoted by annotating the corresponding graph item with a variable name.

We introduce the only flattening axiom $\mathsf{flat_P}$ into $\equiv_{\mathrm{D}}$, and extrusion axioms $\mathsf{extr_S}$, $\mathsf{extr_D}$, $\mathsf{extr_I}$, $\mathsf{extr_F}$. 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). The explicit embedding of sessions is not strictly necessary but it provides an intuitive visual representation. As usual, flattening processes allows for getting rid of the axioms for parallel composition (see [18]). The presence of extrusion

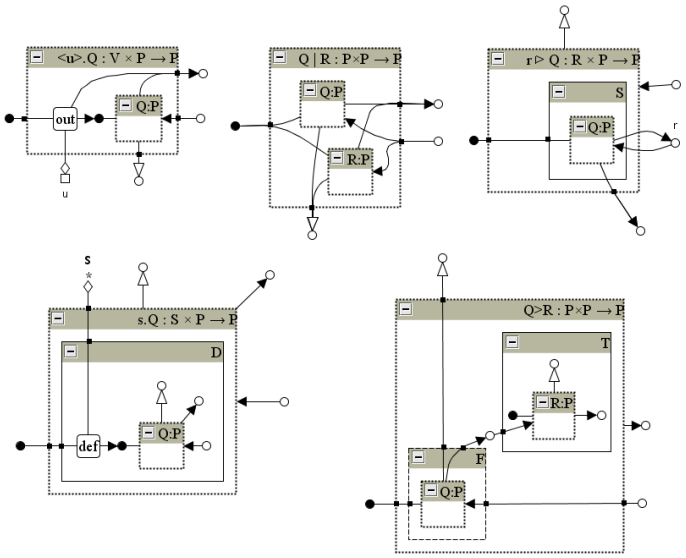

Figure 13: Graphical interpretation for CaSPiS.

axioms is motivated by the structural congruence axioms of CaSPiS, namely CA7 motivates $\mathsf{extr}_{\mathsf{F}}$, CA8 motivates both $\mathsf{extr}_{\mathsf{D}}$ and $\mathsf{extr}_{\mathsf{I}}$, and CA9 motivates $\mathsf{extr}_{\mathsf{S}}$. Note that we use dashed border for designs for which the extrusion axiom hold, while designs to be flattened are depicted with dashed borders.

We explain just a few representative operations in detail. The session operations are interpreted as graph operations that wrap a process into a hierarchical S-typed graph which exposes the control point and a return channel. The first is associated to the control point of the resulting P-typed design, while the second is connected to its output channel. Note how session embedding hides the input and output channels of the embedded process: they are connected directly to the dedicated inter-communication node of the session. Another interesting operation is the pipeline. Here, the source and target processes of the pipeline are embedded in F- and T-typed designs. It is worth noting how the input and output channels of each process are connected in a complementary way. The target process hides its control point and communication channels to denote that it is a non-active process. When the source of the pipe is ready to send a value, a copy of the target process is created and the control and channel nodes are connected as expected. A main difference w.r.t. the encoding we provided in [5], where the extrusion

axiom was considered to hold implicitly for all the edges, is that it is no longer necessary to retain the target-pipe operator parametric w.r.t. the free names of the enclosed process: this was necessary in [5] to keep distinct the (non-congruent) CaSPiS processes $(\nu w)(Q > R)$ and $Q > (\nu w)R$ when $w \in fn(R)$, but now their corresponding graphs are clearly distinct because in the former case (νw) appears above the T-typed edge, while in the latter case (νw) appears below the T-typed edge.

Note that the encoding we adopted for the pipe operators actually suggests how to overcome the restriction to their finite fragment for those calculi we presented. Indeed, dealing with replication operators is by no means difficult, by exploiting the hierarchical structure. Of course, the axiom $!P \equiv !P \mid P$ would not hold, since the two terms would have different graphical encoding. However, it would suffice to introduce an unfolding operation, exactly as it happens for the encoding of pipe operators in CaSPiS.

Example 10. Recall the typical session nesting situation presented in Example 8. Figure 14 depicts the graphical representation of our example, where the graph has been 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.

Example 11. Recall the typical pipeline situation presented in Example 9. Its graphical representation is presented in Figure 15 and highlights various aspects of interest: the flow of the information via the input and output channels, the fact that P_2 and P_3 are *inactive* protocols, and the pipe nesting. Since $>$ is not associative $P_1 > (P_2 > P_3)$ and $(P_1 > P_2) > P_3$ are not structurally equivalent and this is faithfully reflected in the graphs.

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

Proposition 6 (correctness of CaSPiS encoding) *For any $Q, R \in \mathcal{P}$ we have $P \equiv_{\mathrm{C}} Q$ iff $P \equiv_{\mathrm{D}} Q$.*

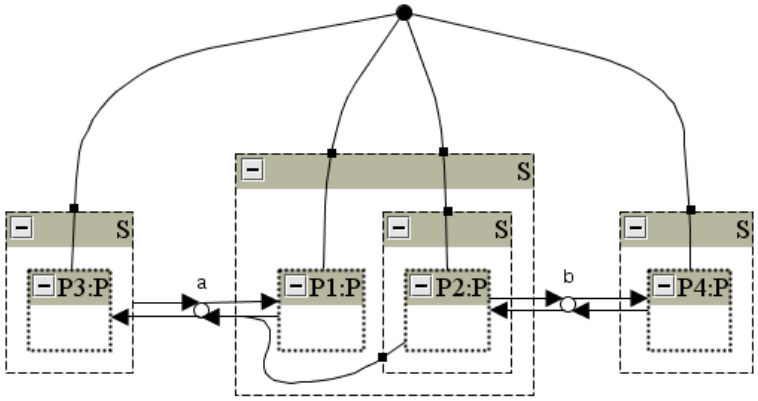

Figure 14: Example of session nesting.

Proof: Soundness of our encoding is reduced to show that for each axiom of $\equiv_{\mathrm{D}}$ (see Definition 21) we have that the left- and right-hand sides are interpreted as equivalent designs terms (according to $\equiv_{\mathrm{D}}$). The proof for AC1 axioms for parallel and non-deterministic composition of processes in $\equiv_{\mathrm{C}}$ is straightforward as we have similar axioms for parallel composition of graphs in $\equiv_{\mathrm{D}}$. A similar reasoning can be applied for the axioms regarding the order of name restrictions and the restriction of 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 similarly shown.

The proof of completeness is along the line of the one provided for π-calculus: taken $P \not\equiv_{\mathrm{C}} P'$ we want to show that $P \not\equiv_{\mathrm{D}} P'$. The normal form of CaSPiS processes is $(\nu W)(\Pi_i R_i > Q_i \mid \Pi_j r_j \triangleright S_j \mid \Pi_k A_k.M_k)$, where each name in W is used at least once and each R_i, Q_i, S_j, M_k is also in normal form. If P and P' have the same outermost shape then a simple inductive argument allows us to conclude that $P \not\equiv_{\mathrm{D}} P'$. If they have different shapes, then we compare all the possibilities for this to happen to conclude that $P \not\equiv_{\mathrm{D}} P'$. The comparison can be carried out similarly to the case of π-calculus: it is rather long because several cases must be considered, but not particularly difficult, because the encoding of each construct (besides

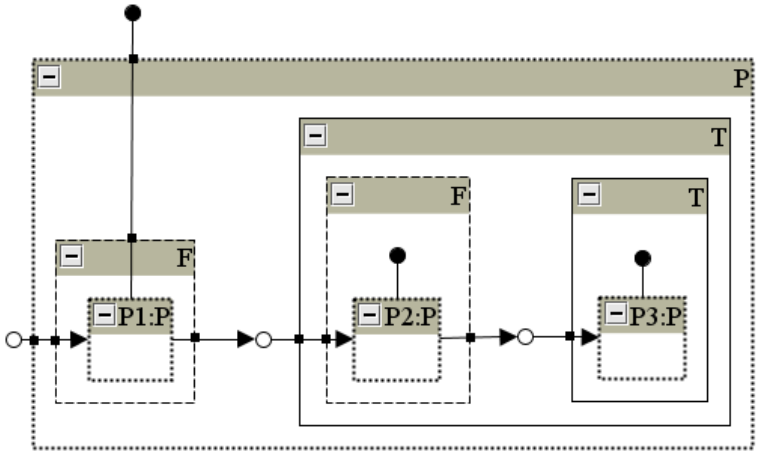

Figure 15: Example of pipelining.

ordinary parallel composition and restriction) introduces an edge or a design that will not be flattened. □

8 Related Work

On the algebra of graphs. Our most direct source of inspiration is an approach for the reconfiguration of software architectures called *Architectural Design Rewriting* (ADR) [7], where architectures are encoded as terms of a particular graph algebra and reconfigurations are defined using standard term rewriting techniques. Our model of hierarchical graphs extends ADR graphs with node sharing and our algebra equips ADR with a suitable syntax. In particular, original ADR specifications can be seen as rewrite theories over a signature formed by derived operations defined by terms closed with respect to nodes. Our algebra, hence, inherits the characteristics of ADR, like the ability to nicely model style-preserving architectural reconfigurations [7].

Our syntax is inspired by the graph algebra proposed in [12]. The main idea there was to have constructors such as the empty graph, single edges, and parallel composition, and axioms like associativity and commutativity of such composition, in order to consider graphs up to isomorphism. Our richer design algebra includes hierarchical features and it is intended to enable a

more suitable representation for nominal calculi and their behaviour.

Concerning set-theoretical formalisms, a direct reference is the framework for hierarchical graph transformation introduced in [14], of which our proposal can be considered an extension, dealing with free names, along the lines of so-called graphs with interfaces discussed in e.g. [18]. Indeed, as far as the mapping of processes is concerned, our solution follows closely [18]: the operators verifying the AC1 axioms basically disappear, while name restriction is dealt with by handling the interfaces. The encoding in [18] actually deals with flat graphs, which suffices for the finite fragment of the calculus. It is however noteworthy that, for the finite fragment, the two proposal coincide (process are mapped into isomorphic graphs by the two encodings). Other set-theoretical models of hierarchical graphs exist in the line of [14] (e.g. [10, 24]), but most of them lack an algebraic syntax and an associated set of axioms.

On structured graphical models. Our approach is closely related to other formalisms that adopt a graphical representation of concurrent systems. Among those, we mention Synchronized Hyperedge Replacement (SHR) [17] and Bigraphical Reactive Systems (BRSs) [23].

The syntax of SHR is basically the one of [12], and it is subsumed by our algebra. Instead, the SHR approach focuses on the description of the operational behaviour of a system by a set of suitably labelled inference rules, which may involve complex synchronisations. We discuss later some of the rewriting features we intend to add to our approach. However, we can safely say that so far the concerns of the two proposals have been orthogonal.

A bigraph is given by the superposition of two independent graphs, representing the locality and the connectivity structure of a system, respectively. In our terms, the first specifies the hierarchical structure of the system, while the second the naming topology. We believe that the two approaches have the same expressiveness, but argue for the better usability of our syntax and the small, intuitive set of axioms. Most importantly, BRSs have been mostly studied in connection with the relative pushout (RPO) technique [21], in order to distill a bisimilarity congruence from a set of rewrite rules. Our hierarchical graphs form a category with pushouts (indeed, possibly an adhesive one), and the DPO approach could be then lifted, as in [14]. Hence, they should be amenable to the borrowed context technique for distilling RPOs [16]. Our proposal thus fits in the standard graph-theoretic mold, while its slender syntax provide a simple intermediate language between

process calculi and their graphical models. Obviously, a possible integration is to use our syntax in order to characterise certain classes of bigraphs (e.g. *pure* bigraphs). Such an integration is suggested in [20], where the authors propose an algebraic syntax for denoting bigraphs and present type systems to characterise those terms that correspond to particular sub-classes.

On rewriting mechanisms. Concerning the operational behaviour of our specifications, we would like to find a term rewriting-like technique for the reconfiguration of designs, and prove it compatible with a graph theoretical approach for rewriting hierarchical graphs. In other words, the correspondence holding between designs and hierarchical graphs should be lifted at the level of rewriting. The standard notions of term rewriting can be applied to our algebra of designs, simply considering sets of (name and design) variables. The corresponding technique for graph rewriting is more complex, since most of these techniques are eminently local, thus making it difficult to simulate the replication of an unspecified design. Nevertheless, since our category admits pushouts, a clear path is laid down by the use of rule schemata in the DPO approach, as in [14].

9 Conclusions

We introduced a novel 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 scopes and containments (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. Indeed, the main result of the paper already guarantees that equivalent terms correspond to isomorphic graphs.

Summing up, we believe that our approach can serve as an inspiration to equip well-known graphical models of communication with syntactical notations that facilitate the definition of intuitive and correct encodings of structured specifications, such as those obtained by using process calculi.

Applications. We are applying our technique to various languages, focusing on process calculi exhibiting nested features. A preliminary proof of the flexibility of our approach for this purpose is found in [5]. Another focus is on metamodels: we plan to develop a technique to distill algebraic specifications out of MOF metamodels, along the lines of [3] but capturing composition as nesting. Some preliminary results in this direction are in [2].

An implementation of our approach and its integration in our prototypical implementation of ADR [6] in the rewrite engine Maude is under current work. A preliminary version is available (at `http://www.albertolluch.com/adr2graphs/`) as a visualiser that considers our design algebra and encodings of process calculi like the π-calculus and CaSPiS, among others.

Acknowledgements

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

References

[1] 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.

[2] A. Boronat, R. Bruni, A. Lluch Lafuente, U. Montanari, and G. Paolillo. Exploiting the hierarchical structure of rule-based specifications for decision planning. In J. Hatcliff and E. Zucca, editors, *Proceedings of the IFIP International Conference on Formal Techniques for Distributed Systems (FMOODS/FORTE'10)*, volume 6117 of *Lecture Notes in Computer Science*, pages 2–16. Springer Verlag, 2010.

[3] A. Boronat and J. Meseguer. An algebraic semantics for MOF. In J. Fiadeiro and P. Inverardi, editors, *Proceedings of the 11th International Conference on Fundamental Aspects of Software Engineering (FASE'08)*, volume 4961 of *Lecture Notes in Theoretical Computer Science*, pages 377–391. Springer Verlag, 2008.

[4] R. Bruni, F. Gadducci, and A. Lluch Lafuente. An algebra of hierarchical graphs. In M. Hofmann and M. Wirsing, editors, *Proceedings of the 5th International Symposium on Trustworthy Global Computing (TGC'10)*, Lecture Notes in Computer Science. Springer Verlag, 2010. To appear.

[5] R. Bruni, F. Gadducci, and A. Lluch Lafuente. A graph syntax for processes and services. In J. Su and C. Laneve, editors, *Proceedings of the 6th International Workshop on Web Services and Formal Methods (WS-FM'09)*, volume 6194 of *Lecture Notes in Computer Science*, pages 46–60. Springer Verlag, 2010.

[6] R. Bruni, A. Lluch Lafuente, and U. Montanari. Hierarchical design rewriting with Maude. In G. Rosu, editor, *Proceedings of the 7th International Workshop on Rewriting Logic and its Applications (WRLA'08)*, volume 238 (3) of *Electronic Notes in Theoretical Computer Science*, pages 45–62. Elsevier, 2009.

[7] 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.

[8] 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.

[9] 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.

[10] G. Busatto, H.-J. Kreowski, and S. Kuske. Abstract hierarchical graph transformation. *Mathematical Structures in Computer Science*, 15(4):773–819, 2005.

[11] A. Corradini and F. Gadducci. An algebraic presentation of term graphs, via gs-monoidal categories. *Applied Categorical Structures*, 7(4):299–311, 1999.

[12] A. Corradini, U. Montanari, and F. Rossi. An abstract machine for concurrent modular systems: CHARM. *Theoretical Computer Science*, 122(1-2):165–200, 1994.

[13] 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 Transformations, Volume 1: Foundations*, pages 163–246. World Scientific, 1997.

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

[15] 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.

[16] H. Ehrig and B. König. Deriving bisimulation congruences in the DPO approach to graph rewriting with borrowed contexts. *Mathematical Structures in Computer Science*, 16(6):1133–1163, 2006.

[17] 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 *Lecture Notes in Computer Science*, pages 22–43. Springer Verlag, 2006.

[18] 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.

[19] F. Gadducci and G. V. Monreale. A decentralized implementation of mobile ambients. In H. Ehrig, R. Heckel, G. Rozenberg, and G. Taentzer, editors, *Proceedings of the 4th International Conference on Graph Transformation (ICGT'08)*, volume 5214 of *Lecture Notes in Computer Science*, pages 115–130. Springer, 2008.

[20] D. Grohmann and M. Miculan. Graph algebras for bigraphs. In J. Küster and E. Tuosto, editors, *Proceedings of the 10th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT'10)*, Electronic Communications of the EASST, 2010. To appear.

[21] J. J. Leifer and R. Milner. Deriving bisimulation congruences for reactive systems. In C. Palamidessi, editor, *Proceedings of the 11th International Conference on Concurrency Theory (CONCUR'00)*, volume 1877 of *Lecture Notes in Computer Science*, pages 243–258. Springer Verlag, 2000.

[22] R. Milner. *Communicating and Mobile Systems: The π-calculus*. Cambridge University Press, 1992.

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

[24] W. Palacz. Algebraic hierarchical graph transformation. *Journal of Computer and System Sciences*, 68(3):497–520, 2004.

[25] G. Paun. *Membrane Computing. An Introduction*. Springer, 2002.

[26] G. Paun and G. Rozenberg. A guide to membrane computing. *Theoretical Computer Science*, 287(1):73–100, 2002.