An Algebra of Hierarchical Graphs *

Roberto Bruni[1], Fabio Gadducci[1], and Alberto Lluch Lafuente[2]

[1] Department of Computer Science, University of Pisa, Italy
[2] IMT Institute for Advanced Studies Lucca, Italy

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.

1 Introduction

As witnessed by a vast literature, graphs offer a convenient ground for the specification and analysis of software systems. Roughly, graphical models expose the structure of a system in terms of its computational components, their ports and their connectivity. Using *plain hypergraphs* (i.e. graphs where nodes and edges form just sets, with no additional structure), components and connectors become hyperedges and their ports become nodes. Moreover, nodes, hyperedges and their tentacles can be *typed* so to discard erroneously linked systems.

In [1] we argue that *structured* graphs are most suited for service-oriented systems, where scalable techniques and open-ended specifications are important issues that are not immediately met by plain hypergraphs alone. Structured graphs offer better support for "understanding" graphs (like parsing and browsing large systems), for designing systems (like expressing requirements and specifications, facilitating abstraction and refinement, allowing modularity and seamless aggregation), supporting automated analysis and verification (like model construction, model conformance, behavioural analysis, assessing sound reconfiguration and refactoring transformations) and last but not least, for sound and complete visual encoding of computational systems.

Different kinds of structures can be super-imposed on graphs. First, a graph G can be enclosed in some sort of box whose label L implicitly defines some properties of the enclosed graph, i.e., its *style* (e.g. see the graph transformation framework in [2]). Figure 1 (left) shows one example of "topologically" labelled graph, that can be written, e.g., $Seq[G]$ (for the obvious plain graph G derivable from the figure) or, equivalently, as a membership annotation $G : Seq$, where Seq can be read as the set of all (well-linked) sequential graphs.

The "graphs within boxes" view can be enhanced into a "graphs within edges" view, where boxes have their own tentacles and boxing can be iterated.

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

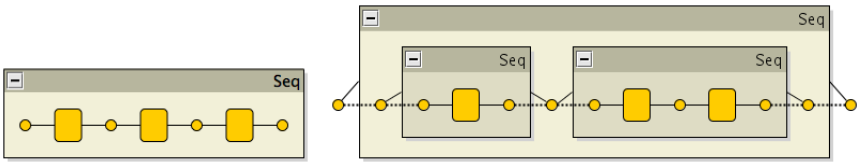

Fig. 1. Graphs within boxes (left) and graphs within edges (right) views

For example, Fig. 1 (right) shows that the sequential composition of sequential graphs still yields a sequential graph. Note that the boxed interfaces are equipped with tentacles and dotted lines make explicit the link between inner nodes, exposed by interfaces, and actual nodes (analogous notation will be used in Fig. 6 (right) and Fig. 7). This way, boxes can be read as enhanced interfaces allowing for more sophisticated forms of containment, (well-typed) composition, modular specification, logical hierarchies or node sharing.

The encoding of configurations given with an algebraic specification language (e.g. as in process calculi) is best defined by structural induction. In absence of an algebraic presentation for the target model, an ad-hoc algebraic syntax must be developed in order to benefit from structural induction in proofs, transformations or definitions. An example of this is the algebraic presentation of MOF (Meta Object Facility) metamodels of [3]. Still, most graph models are not equipped with an algebraic syntax and those that exist require advanced skills to deal with sophisticated models involving set-theoretic definitions of graphs with interfaces (e.g. [4]) or complex type systems (e.g. [5]), hampering definitions and proofs. Moreover, one encounters a severe drawback: namely, the syntax of a graph formalism is often very different from the source language and not provided with suitable primitives to deal with features that commonly arise in algebraic specifications, like names (e.g. references, channels), name restrictions (e.g. hiding, nonce generation) or hierarchical aspects (e.g. ambients, scopes) in the case of process calculi. Additionally, any graphical encoding involves the challenge of preserving structural equivalence of system configurations, i.e. ensuring that structurally equivalent configurations are mapped to isomorphic graphs. For example, in graph transformation approaches [6] the soundness of the encoding is necessary to model dynamic aspects like operational semantics, reconfigurations, refactorings or model transformations, because the matching of redexes is based on (sub)graph isomorphism.

In order to overcome such challenges, we have developed a handy syntax for representing nested graphs and reducing the representation distance w.r.t. specification languages. The syntax has been first presented in [7] together with a methodology to encode process calculi like, among other case studies, a sophisticated calculus for the description of service-oriented applications, CaSPiS [8], whose features posed further challenges to visualisation, due to the interplay of name handling, nested sessions and a pipeline operator.

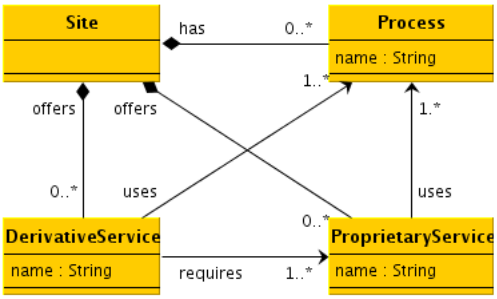

Fig. 2. SPS metamodel for our running scenario

The contribution of this paper is to equip the syntax with a set of axioms and a suitable domain of interpretation, thus resulting in a novel algebra of hierarchical graphs. The domain of interpretation is a (sound and complete) initial model that serves as an original flavour of (layered) graphs: the axioms allow for term normalization and the interpretation of terms over set-theoretical models allow us to use the algebra as some sort of intermediate language, reducing the representation distance between specification languages and structured graph models. This paper, hence, is the foundational counterpart of our methodological approach to the visual specification of systems initiated in [7].

Synopsis. We take a scenario based on a simple metamodel of service oriented entities as a running example, introduced in § 2. The algebra of nested graphs is defined in § 3 and its set-theoretical interpretation is defined in § 4, together with the main result establishing the soundness and completeness of the interpretation. Finally, related and future works are discussed in § 5 together with some concluding remarks. A short appendix addresses a practical issue, raised in [7], concerning the possibility of flattening certain layers of a hierarchical graph.

2 Sites, processes and services

Nesting and linking are two key structural aspects that arise repeatedly in computer systems: consider e.g. the structure of file systems, composite diagrams, networks, membranes, sessions, transactions, locations, structured state machines or XML files. Identifying the right structure and level of abstraction is fundamental to enjoy scalability. In particular, nesting (called composition in MOF) plays a fundamental role for abstracting the complexity of a system by offering different levels of detail. We argue that nesting and linking must be treated as first-class concepts, conveniently represented with a suitable syntax that allows one to express and exploit them. Various graphical models of nesting and sharing structures already exist but (as we argue in § 5) it seems to us that none of them offers a syntax as simple and intuitive as the one proposed in this paper.

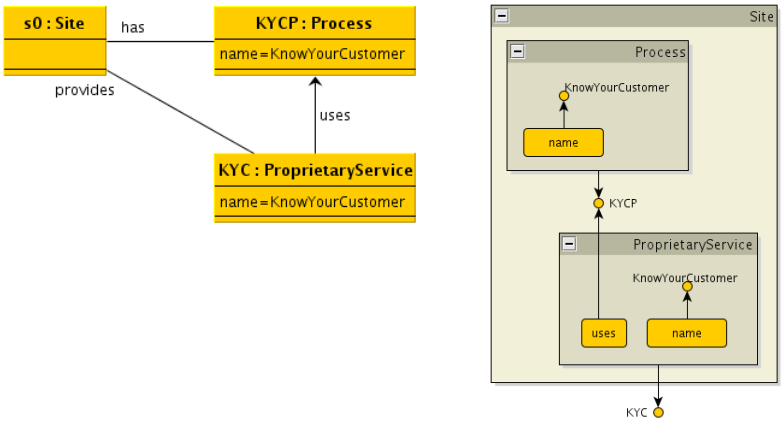

Fig. 3. An SPS instance as a flat diagram (left) and as a nested graph (right)

As a simple running example, we consider the metamodel SPS (for sites, processes and services) shown in Fig 2. It fixes an alphabet of (attributed) entities (sites, processes, derivative and proprietary services) and their possible relations: processes and services are associated to sites (containment is given by *composition relations*, which are denoted by lines decorated with diamonds on the container end), proprietary services may *use* processes, and derivative services may also *require* services (*association relations* are denoted with ordinary arrows). The algebraic presentation of [3] basically consists of representing models as multisets of (typed) objects with some attributes used for their interrelations (i.e. references to object identifiers) Roughly, each configuration (object multiset) corresponds to a flat graph where nodes and edges are used to represent objects and their relations as depicted in the example instance of Fig. 3 (left), where a site provides a client certification service (Know Your Customer) built out from an internal process. Composition is represented just as any other relation (the *has* and *provides* relations) which makes it difficult to exploit the compositional structure to abstract or manipulate such models. For instance, it is not easy to write a term $\mathbf{site}(x)$ that matches a site with any possible configuration x of processes and services because the multiset representation requires us to see the configuration as $\mathbf{site}(x), C$, where $\mathbf{site}(x)$ is a configuration containing the process and all its contents for which we need to check that C (the rest of the configuration) does not contain any object referring to the site as its container. Matching a term like $\mathbf{site}(x)$ in a graphical representation would mean to match an entire subgraph which is clearly facilitated when graphs are structured (e.g. hierarchical). For instance, the graph on Fig. 3 (right) offers an explicit, visual representation of composition by containment. Now, $\mathbf{site}(x)$ can be used to denote a *Site*-labelled box embedding x (the content of the site).

3 An algebra of hierarchical graphs

We introduce here our algebra of (typed) hierarchical graphs with edge-like interfaces that we call *designs*. The algebraic presentation of designs has emerged during our studies on *Architectural Design Rewriting* [9] (hence the name) and it has been inspired by the graph algebra of [10].

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

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

where l *and* L *are drawn from 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.*

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

Terms generated by $\mathbb{G}$ and $\mathbb{D}$ are meant to represent (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 $fn(\cdot)$

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

Example 1. Let $a, b \in \mathcal{E}$, $A \in \mathcal{D}$, $u, v, w, x, y \in \mathcal{N}$. We write and depict in Fig. 4 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

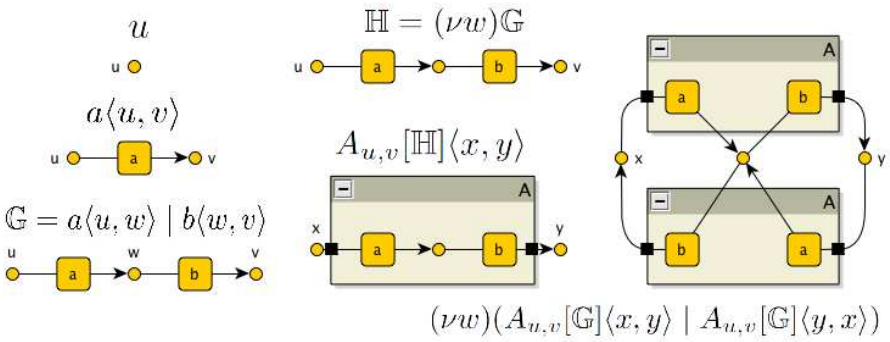

Fig. 4. Some terms of the graph algebra

with black boxes on design borders. In the particular examples only free nodes are annotated with their identities. Note that this representation is informal (alike Fig. 3 (right) and Fig. 5) 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 2 (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 $\lfloor\overline{x}\rfloor \subseteq fn(\mathbb{G})$; and (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 [10] 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 local (DA10).

Definition 3 (design axioms). *The structural congruence $\equiv_{\mathrm{D}}$ over well-formed designs and graphs is the least congruence satisfying*

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

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

Note that $\equiv_{\mathrm{D}}$ respects free nodes, i.e. $\mathbb{G} \equiv_{\mathrm{D}} \mathbb{H}$ implies $\mathit{fn}(\mathbb{G}) = \mathit{fn}(\mathbb{H})$. Being $\equiv_{\mathrm{D}}$ a congruence, we remark e.g. that $L_{\overline{x}}[\mathbb{G}] \equiv_{\mathrm{D}} L_{\overline{x}}[\mathbb{H}]$ whenever $\mathbb{G} \equiv_{\mathrm{D}} \mathbb{H}$.

In the following, we shall often write $L[\mathbb{G}]\langle \overline{y} \rangle$ as a shorthand for $L_{\overline{y}}[\mathbb{G}]\langle \overline{y} \rangle$.

Example 2. Recall the example of Section 2 and consider the graph on the right of Fig. 3. Its syntactical representation **site0** is defined as

Site[(νKYCP)(Process[KYCP | **attr**(name, KnowYourCustomer)]⟨KYCP⟩
| PropietaryService[KYC | **attr**(name, KnowYourCustomer) | uses⟨KYCP⟩]⟨KYC⟩)]

where $\mathbf{attr}(l, y) \stackrel{\mathbf{def}}{=} (\nu y)l\langle y \rangle$ is an abbreviation for the representation of an attribute as an edge with the name of the attribute as label attached to a new node representing the value. Note that other representations can be chosen for a pure graphical representation of attributes (for instance sharing values).

The syntactical presentation is very compact and clean. Note for instance how some structural constraints are captured: the impossibility for a service to use a process of another site (an OCL constraint not shown in Fig 2 for brevity) is ensured by the restriction of the identity of processes inside sites. Now, recall the convenience of being able to express a term like $\mathbf{site}(x)$. In our syntax we can define $\mathbf{site}(x) \stackrel{\mathbf{def}}{=} \mathrm{Site}[x]$, i.e., a Site-labelled design with some graph x in it. Clearly, our **site0** matches $\mathbf{site}(x)$ with x being the graph representing processes and services of the site. We can then perform some proof based on induction on the compositional structure of sites but also define rewrite rules like $\mathbf{site}(x) \mid \mathbf{site}(y) \to \mathbf{site}(x \mid y)$ for fusing two sites, which would require a cumbersome set of rules when working with plain graphs or multisets.

Let us now consider the more complex instance of Fig. 5, with some non-trivial linking modelling the fact that a derivative service (a certified mini credit service) is built using an external client certification service (KYC). The term underlying the graphical representation is **site0** | **site1** where **site1** is defined as

Site[(νMCP)(Process[MCP | **attr**(name, MiniCredit)]⟨MCP⟩
| DerivativeService[CMC | **attr**(name, CertifiedMiniCredit)
| uses⟨MCP⟩ | requires⟨KYC⟩]⟨CMC⟩
| ProprietaryService[MC | **attr**(name, UncertifiedMiniCredit) | uses⟨MCP⟩]⟨MC⟩)]

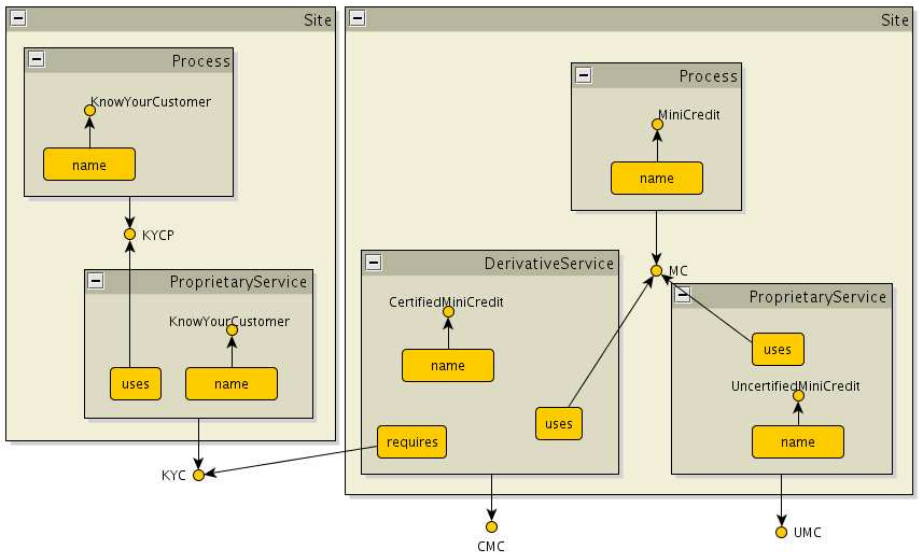

Fig. 5. An instance of the SPS metamodel as a hierarchical graph

It is worth to observe how the model is structured by the graph, hiding the processes inside sites and allowing for cross-references to services only, as in the case of the derivative service CMC of **site1** that requires the proprietary service KYC of **site0**.

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

Definition 4 (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 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 $fn(L^i_{\overline{y}_i}[\mathbb{G}_i]\langle \overline{w}_i \rangle) = Z$ for all $i = 1...q$.

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

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 is actually proved to be unique.

4 A model of hierarchical graphs

The family 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 5 (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 6 (hierarchical graph). *The set $\mathcal{H}$ of* hierarchical graphs *is the smallest set*[3] *containing all the tuples $G = \langle N_G, E_G, t_G, i_G, x_G, r_G, F_G\rangle$ where*

1. *$\langle N_G, E_G, t_G, F_G\rangle$ 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 [2], 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 [4]. 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.

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

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

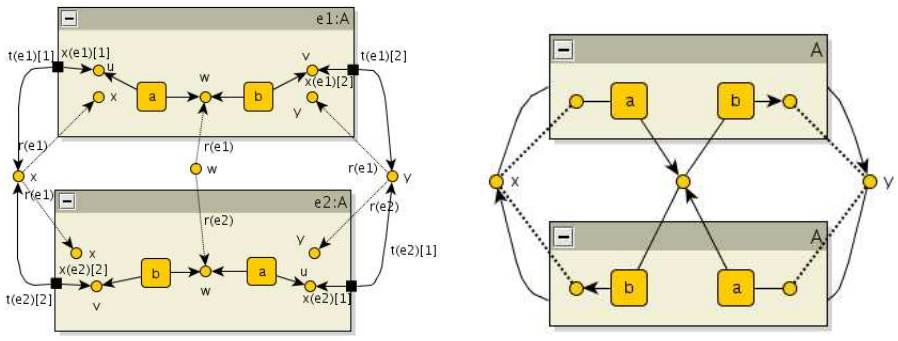

Fig. 6. A hierarchical graph (left) and its simplified representation (right)

An intuitive way to understand our model is a programming metaphor where each hierarchical edge e is 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 Fig. 4 (right). Its actual interpretation as a hierarchical graph appears in Fig. 6 (left) decorated with the most relevant annotations (the tentacle, exposition and renaming functions for the two hierarchical edges). As witnessed by Fig. 6 (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 Fig. 1 (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 Fig. 4 and Fig. 5.

Example 4. Recall our example of services and the instance with two sites for which we gave its syntactical representation as **site0** | **site1** (see Example 2) and its informal graphical representation (see Fig. 5). Its actual hierarchical graph is depicted in Fig. 7 where we do not offer all the annotations as in the previous example and we hide some useless copies of global nodes (just to allow the reader to focus on the relevant part of the example). To a certain extent, it might be argued that our formal model is redundant, in the sense that global nodes require a copy at each subgraph. As we will see, this is necessary for the completeness result. In the informal presentation, as well as e.g. in a visualising tool, all copies are put together at the intuitively "right" level.

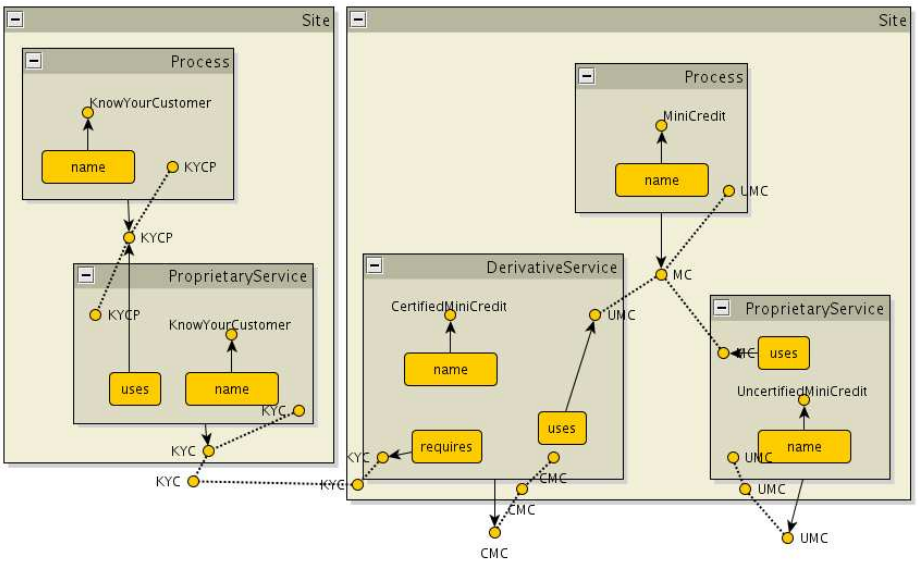

Fig. 7. An instance of our SPS metamodel as a formal hierarchical graph

These examples should hopefully outline how our model of hierarchical graphs works and the comparison with the informal representation should suggest how they could be used to obtain an intuitive, clear visualisation. The examples should also highlight that the algebra is providing a simple syntax that hides the complexities of hierarchical models. 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 *height* (resp. *depth* or *layer*) of a graph is the height (resp. depth) of its embedding tree. The leaves of the embedding tree are actually plain graphs. 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 7 (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;*

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

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 form one can define. In particular, using the terminology of [11] they are *root-level* in the sense that they represent a layer-by-layer embedding. More general notions are the *deep* morphisms of [11] 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, which we use to define a composition operator and which prepare the ground for some basic pushout-based graph transformations.

Proposition 2 (pushouts [6]). *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.

Encoding terms into graphs. The last step before introducing the algebraic characterisation of graphs is the definition of a composition operator. We need however a few auxiliary definitions.

Definition 8. *Let $N \in \mathcal{N}$ be a subset of nodes of graph G. Then, $\widehat{N}$ is the hierachical 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.*

We denote the empty function with $\bot$, distinguishing it from the empty set $\emptyset$.

Definition 9 (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 $\widehat{F_G \cap F_H} \to H$.*

Graph composition is always defined, thanks to Proposition 2. 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 10 (graph interpretation). *The encoding $[\![\cdot]\!]$, mapping well-formed terms into graphs, is the function inductively defined as*

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

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.

It is worth to remark that the 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 $[\![\mathbb{G}]\!]$.*

Moreover, 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 $[\![\mathbb{G}_1]\!]$ is isomorphic to $[\![\mathbb{G}_2]\!]$.*

The proof proceeds by exploiting the normalized form of well-formed terms. In fact, by Prop. 3 each graph has associated a well-formed term in normal form, and this can be further exploited to prove the uniqueness of such term.

5 Conclusions and Related and Future Works

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 ranging from process calculi with notion of scopes and containments (like ambients, membranes, sessions and transactions) to metamodels with composition relations. Some advantages of our approach are due to the graph algebra. Most importantly, its 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, i.e. to facilitate proofs by structural induction. Indeed, the main result of the paper already guarantees that equivalent terms correspond to isomorphic graphs.

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) [9], 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 [9].

Our syntax is inspired by the graph algebra proposed in [10]. 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. A key difference is that in our initial model, a node restriction cannot cross the boundaries of hierarchical edges in which it is contained. Adding the corresponding axiom is feasible, even if it would result in a quite different set-theoretic notion of hierarchical graph. A less demanding, yet quite useful alternative is linked with the possibility of "flattening" some of the designs, in order to consider them just as type annotations Accomodating for these axioms, fruitfully used in [7], would not change our class of hierarchical class, as shown in the Appendix of the present paper.

Concerning set-theoretical formalisms, a direct reference is the framework for hierarchical graph transformation introduced in [2], of which our proposal can be considered an extension, dealing with free names, along the lines of so-called graph with interfaces discussed in e.g. [4]. Indeed, as far as the mapping of processes is concerned, our solution follows closely [4]: the operators verifying the AC1 axioms basically disappear, while name restriction is dealt with by handling the interfaces. Other models of hierarchical graphs exist in the line of [2] (e.g. [12, 11]), 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 Bigraphical Reactive Systems (BRSs) [13] and Synchronized Hyperedge Replacement (SHR) [14].

The syntax of SHR is basically the one of [10], 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 largely 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 [15], 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 [2]. 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 [17], 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 [2].

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 [7], offering an encoding of a session-centered calculus. 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.

An implementation of our approach and its integration in our prototypical implementation of ADR [18] 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 some encodings of process calculi like the π-calculus and CaSPiS.

Acknowledgements. We are grateful to Andrea Corradini for his many suggestions and to Artur Boronat for fruitful discussions.

References

1. Bruni, R., Lluch Lafuente, A.: Ten virtues of structured graphs. In Boronat, A., Heckel, R., eds.: Proceedings of the 8th International Workhshop on Graph Transformation and Visual Modeling Technique (GT-VMT'09). Volume 18 of Electronic Communications of the EASST., ECEASST (2009)
2. Drewes, F., Hoffmann, B., Plump, D.: Hierarchical graph transformation. Journal on Computer and System Sciences **64** (2002) 249–283
3. Boronat, A., Meseguer, J.: An algebraic semantics for MOF. In Fiadeiro, J., Inverardi, P., eds.: Proceedings of the International Conference on Fundamental Aspects of Software Engineering (FASE'08). Volume 4961 of Lecture Notes in Theoretical Computer Science., Springer Verlag (2008) 377–391
4. Gadducci, F.: Term graph rewriting for the pi-calculus. In Ohori, A., ed.: Proceedings of the 1st Asian Symposium on Programming Languages and Systems (APLAS'03). Volume 2895 of Lecture Notes in Computer Science., Springer Verlag (2003) 37–54

5. Bundgaard, M., Sassone, V.: Typed polyadic pi-calculus in bigraphs. In Bossi, A., Maher, M.J., eds.: Proceedings of the 8th International Symposium on Principles and Practice of Declarative Programming (PPDP'06), ACM (2006) 1–12
6. Corradini, A., Montanari, U., Rossi, F., Ehrig, H., Heckel, R., Löwe, M.: Algebraic Approaches to Graph Transformation - Part I: Basic Concepts and Double Pushout Approach. In Rozenberg, G., ed.: Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations, World Scientific (1997) 163–246
7. Bruni, R., Gadducci, F., Lluch Lafuente, A.: A graph syntax for processes and services. In Jianwen, S., Laneve, C., eds.: Proceedings of the 6th International Workshop on Web Services and Formal Methods (WS-FM'09). Lecture Notes in Computer Science, Springer Verlag (2009) To appear.
8. Boreale, M., Bruni, R., De Nicola, R., Loreti, M.: Sessions and pipelines for structured service programming. In Barthe, G., de Boer, F.S., eds.: 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., Springer Verlag (2008) 19–38
9. Bruni, R., Lluch Lafuente, A., Montanari, U., Tuosto, E.: Style Based Architectural Reconfigurations. Bulletin of the European Association for Theoretical Computer Science (EATCS) **94** (2008) 161–180
10. Corradini, A., Montanari, U., Rossi, F.: An abstract machine for concurrent modular systems: CHARM. Theoretical Computer Science **122** (1994) 165–200
11. Palacz, W.: Algebraic hierarchical graph transformation. Journal of Computer and System Sciences **68** (2004) 497–520
12. Busatto, G., Kreowski, H.J., Kuske, S.: Abstract hierarchical graph transformation. Mathematical Structures in Computer Science **15** (2005) 773–819
13. Milner, R.: Pure bigraphs: Structure and dynamics. Information and Computation **204** (2006) 60–122
14. Ferrari, G.L., Hirsch, D., Lanese, I., Montanari, U., Tuosto, E.: Synchronised hyperedge replacement as a model for service oriented computing. In de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P., eds.: Proceedings of the 4th International Symposium on Formal Methods for Components and Objects (FMCO'05). Volume 4111 of Lecture Notes in Computer Science., Springer Verlag (2006) 22–43
15. Leifer, J.J., Milner, R.: Deriving bisimulation congruences for reactive systems. In Palamidessi, C., ed.: Proceedings of the 11th International Conference on Concurrency Theory (CONCUR'00). Volume 1877 of Lecture Notes in Computer Science., Springer Verlag (2000) 243–258
16. Ehrig, H., König, B.: Deriving bisimulation congruences in the DPO approach to graph rewriting with borrowed contexts. Mathematical Structures in Computer Science **16** (2006) 1133–1163
17. Grohmann, D., Miculan, M.: Graph algebras for bigraphs. In Boronat, A., Heckel, R., eds.: Proceedings of the 10th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT'10). Electronic Communications of the EASST, (ECEASST) To appear.
18. Bruni, R., Lluch Lafuente, A., Montanari, U.: Hierarchical design rewriting with Maude. In Rosu, G., ed.: Proceedings of the 7th International Workshop on Rewriting Logic and its Applications (WRLA'08). Volume 238 (3) of Electronic Notes in Theoretical Computer Science., Elsevier (2009) 45–62
19. Drewes, F., Kreowski, H.J., Habel, A.: Hyperedge replacement graph grammars. In Rozenberg, G., ed.: Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations, World Scientific (1997) 95–162

Appendix: Flattening

We call a graph *flat* whenever there is no design in its body. Flattening a design is done by a kind of hyper-edge replacement [19] in the form of axioms that are sometimes useful to be included in the structural congruence.

Example 5. Suppose that we want to characterise the set of a-labelled, acyclic, and connected sequences (see Example 1). We can define an algebra with an element α in the sequence, and a binary sequential composition $_;_$. Both are derived operators defined by $\alpha \stackrel{\text{def}}{=} A_{(u,v)}[a(u,v)]$ and $X;Y \stackrel{\text{def}}{=} A_{(u,v)}[(\nu w)(X\langle u,w\rangle \mid Y\langle w,v\rangle)]$, where X and Y have type A. Clearly, the algebra as such constructs hierarchical sequences, where e.g. $(\alpha;(\alpha;\alpha))\langle x,y\rangle$ and $((\alpha;\alpha);\alpha)\langle x,y\rangle$ are not equivalent graphs due to different nestings.

Definition 11 (flattening axiom). *Given a design label $L \in \mathcal{D}$, its* flattening *axiom* $\mathsf{flat}_{\mathsf{L}}$ *is* $L_{\overline{x}}[\mathbb{G}]\langle\overline{y}\rangle \equiv \mathbb{G}\{\overline{y}/\overline{x}\}$.

Example 6. By introducing $\mathsf{flat}_{\mathsf{A}}$ in the algebra of Example 5, the two former terms $(\alpha;(\alpha;\alpha))\langle x,y\rangle$ and $((\alpha;\alpha);\alpha)\langle x,y\rangle$ are identified, and correspond to the plain graph $(\nu w_1,w_2)(a(x,w_1) \mid a(w_1,w_2) \mid a(w_2,y))$.

The above example illustrates the two roles of the nesting operator: as a means to enclose a graph and as a sort of typed interface to enable disciplined graph compositions. The presence of flattening axioms makes the first role implicit. The example also illustrates how graphical encodings of existing (algebraic) languages are defined and exploited: the main trick is to see the constructors of the original language as derived operators of the graph algebra.

In the presence of flattening, our main result can be extended just by a minor change in the graph interpretation of Definition 10, by letting

$$\begin{array}{ll} [\![L_{\overline{x}}[\mathbb{G}]\langle\overline{y}\rangle]\!] = \langle N_{\mathbb{G}}, \{e\}, e\mapsto\overline{y}, e\mapsto[\![\mathbb{G}]\!]\oplus[\![\lfloor\overline{y}\rfloor]\!], e\mapsto\overline{x}, e\mapsto id_N, N'\rangle & \text{if } \mathsf{flat}_{\mathsf{L}}\notin\equiv_{\mathrm{D}} \\ [\![L_{\overline{x}}[\mathbb{G}]\langle\overline{y}\rangle]\!] = \langle N_{\mathbb{G}}\{\overline{y}/\overline{x}\}, E_{\mathbb{G}}\{\overline{y}/\overline{x}\}, t_{\mathbb{G}}\{\overline{y}/\overline{x}\}, i_{\mathbb{G}}\{\overline{y}/\overline{x}\}, x_{\mathbb{G}}, r_{\mathbb{G}}, N'\rangle & \text{if } \mathsf{flat}_{\mathsf{L}}\in\equiv_{\mathrm{D}} \end{array}$$

where $e \in \mathcal{A}_{\mathcal{D}}$ and N' abbreviates $(F_{\mathbb{G}} \setminus \lfloor\overline{x}\rfloor) \cup \lfloor\overline{y}\rfloor$.