Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science
ELSEVIER Electronic Notes in Theoretical Computer Science 109 (2004) 97–109
www.elsevier.com/locate/entcs

Shaped Hierarchical Architectural Design [2]

Dan Hirsch Ugo Montanari

Dipartimento di Informatica, Università di Pisa,
Via F. Buonarroti 2, I-56127, Pisa, Italia [1]

Abstract

Architectural design of software systems deals with high level configuration structuring. Checking that a system belongs to a software architecture style (or shape) implies that the architecture is an instance of a structurally defined class. On the other side, hierarchies allow modeling at different levels of detail: subsystems may be represented as single components to abstract structure and behavior. This paper proposes a type-based approach for representing hierarchical SA shapes using types. Typing proofs define a general framework based on inference rules where shape rules and graphs representing systems are type judgements. Hierarchy constructors are defined in the type system to construct proof terms capturing hierarchical structure. We claim that proof terms provide more information than just graphs about the process of constructing systems, and allow the specification of reconfigurations as proof term rewritings. Reconfiguration consistency is obtained as subject reduction: as long as cutting and pasting typing proofs still yields typing proofs, subject reduction is guaranteed. As a possible instantiation of the approach, we present the type system for shapes (and hierarchies) with global references.

Keywords: Software Architecture Styles, Hierarchies, Shapes, Type Systems, System Reconfiguration

1 Introduction

The architectural design of software systems is the activity dealing with high level structuring of configurations and rule definition for the construction of a software product [7]. As an answer to the need of formal notations and tools for architecture-based development, *Architecture Description Languages (ADLs)*, where proposed [12]. Two relevant aspects for the description of

[1] Email: dhirsch@di.unipi.it, ugo@di.unipi.it

[2] Research partially supported by the EU FET – GC Project IST-2001-32747 AGILE and the EC RTN 2-2001-00346 SegraVis (Syntactic and Semantic Integration of Visual Modelling Techniques).

doi:10.1016/j.entcs.2004.02.059

software architectures are style checking and hierarchical composition [12]. The formal approach presented in this paper is centered around these aspects and it is being developed in the context of the EU-project AGILE [1].

Style (or shape) checking implies that an architecture is an instance of some *Software Architecture (SA) Style* that characterizes a class of structures exhibiting a common pattern. A style can be seen as a type for a given architecture. Generalizing the idea of style as a type, from now on we use the name *shape* for the rules used to type systems. This term is taken from work on graph grammars for typing programming language pointer-based data structures, where declarations of types are not enough to describe (and validate) more complex structures, like doubly-linked lists or leaf-connected trees.

This paper proposes an approach for representing hierarchical software architecture shapes using types. Typing proofs define a general framework based on inference rules where shape rules and graphs representing system configurations are represented as type judgements. Therefore, if there is a typing proof for a judgment, then the system is correctly shaped (i.e. typed), where the axioms of the type system are the shaping rules of a style.

An aspect strongly related to shape, and where we can contribute, is *SA reconfiguration*. Reconfiguration has to respect shape, i.e. type. But for design, just observing the actual configuration may not be enough. Instead, observing the steps taken to obtain the final system may provide important information about the process of construction. In this line, we claim that proof terms (i.e., terms of rule names encoding typing proofs) provide more information than just graphs about the process of constructing systems and allow to specify reconfigurations as proof term rewritings. Then, reconfiguration consistency is obtained as subject reduction: as long as cutting and pasting typing proofs still yields typing proofs, subject reduction is guaranteed.

Also, our approach allows the integration of shapes and hierarchies. Hierarchical composition allows to describe systems at different levels of detail. From a general point of view, hierarchical structures are present in many aspects related with system configuration. Large scale system development, global computing, wide area networking, etc., introduce requirements for which it may be useful to represent systems with hierarchical structures. Hierarchical levels can represent nested localities: sites, agents, administrative domains that are located at different places, etc. Hierarchies are present in several areas like process calculi (Ambient Calculus [4]), concurrent system modeling (Bigraphs [13]), UML (e.g. state charts with decomposition and refinement). In our case, hierarchical structure is captured via hierarchy constructors which are similar to basic constants, i.e. only type and name is specified. Then for each hierarchical constructor a standard "symbolic" body is defined as a type

judgment. Hierarchical graphs can be derived in the resulting type system.

In summary, our goal is to cope with modeling problems for SA. The use of type systems and linear representation of structures allow the hope of taking advantage of existing work on programming languages and static type checking. In addition, there is a corresponding graphical view for all elements in the approach. As a possible instantiation of the approach we present a type inference system for shapes with hierarchies and global references. Taking some initial ideas from [8,9], judgments describe graphs and shape rules (graph rewriting rules), hierarchy constructors are typed as rules and architectures are derived by typing proofs.

Related Work: We already mentioned *shapes* for pointer structures. Paper [2] proposes graph-reduction (GR) specifications to specify classes of pointer data structures (shapes). GR rules are just reversed graph grammar productions. In [6] context-free graph grammars are proposed to model *shape types*. Both propose algorithms for testing membership and analyze their complexity problems. These papers deal with shapes in the context of programming languages and some results could be relevant to our approach as well. Related with [6] we mention the work of [11] which was the first to propose context-free graph grammars for SA styles. Work on graph transformation for SA reconfiguration is reported in [15]. They present a two-level approach where a program design language is used to represent states and computations, and an algebraic framework based on categories to represent architectures and reconfigurations. These works are related with the general goal of our approach.

Related with hierarchies, we mention the work of [5] on hierarchical graph transformation. In [5] special hyperedges, called *frames*, are introduced containing hierarchical graphs or variables allowing to define rules that can copy and remove subgraphs in a single step. Each hierarchy level is treated independently and relation between levels is defined as new morphisms. For this, the double-pushout [14] approach for graph transformation is extended to hierarchical graphs by recursively constructing pushouts and morphisms. Also, they present a flattening operation where frames in each level are replaced by their contents. Hierarchy is strict and it does not allow border-crossing. In this line, the work of [10] integrates hierarchy and shapes, where edges and nodes are hierarchical, and shapes (context-free rules) are used to type hierarchical levels. Types are defined for variables that must correspond to a shape. In these works the theory of double-pushout is redefined to handle hierarchy.

These papers integrate hierarchy as new kinds of edges and nodes typed by shapes. In our case we use hierarchy constructors in proof terms instead of requiring additional structure in underlying graphs. The approach is more abstract in the sense that hierarchy and shape are directly recoverable from

typing proofs and indirectly from graphs. Style-preserving graph transformations (i.e., reconfigurations) are achieved in our approach by well typed proof term rewriting rules, while in the other approaches (particularly [11]) reconfiguration must be explicitly proved correct. Also we try to reinvent as little as possible about hierarchical graph transformations and their typing techniques by relying on the large body of concepts developed for ordinary graph rewriting and for the type theory of programming languages. Moreover, our goal is to propose a framework that can be instantiated according to problems of interest for SA while the above papers focus on programming languages.

2 Running Example

As running example we use the *Airport Case Study* [1]. The basic scenario is as follows. In airports there are passengers and planes. Planes run flights, and land and take off, transporting passengers and their luggage between airports. Here we take a simplified view of the example for a clear presentation. We concentrate on classes concerning locations, which define a hierarchy. Countries may contain airports, which can contain planes and passengers.

3 Shaped Hierarchical Systems

This section introduces the approach for representing styles using types. Generalizing the idea of style as a type, we use the name *shape* ([2,6]) as a characterization for a class of graphs (representing instances of a style) which are generated from a set of *rewriting rules over graphs called shape rules* (i.e., rules specifying systems typing). Then, typing proofs define a general framework based on inference rules, which can be instantiated producing different type systems allowing to choose the most convenient solution for some specific design problem. A type system is defined via inference rules over type judgments representing shapes and graphs. Therefore, if there is a typing proof for a judgment representing a system instance, then the system is correctly shaped (i.e. typed), where the axioms of the type system correspond to the shaping rules of the style. Also, we obtain a uniform linear representation as type judgments (including hierarchies), that we considered suitable for analysis, together with a graphical representation suitable for modeling.

3.1 Hypergraphs and Syntactic Judgments

System configurations are represented as hypergraphs [14]. Hyperedges correspond to components or modules and their attachment nodes are their communication ports or connections with other components or modules. A *hyperedge*,

or simply an edge, is an atomic item with a label (from a ranked alphabet $LE = \{LE_n\}_{n=0,1,\ldots}$) and with as many (ordered) tentacles as the rank of its label. A set of *nodes* together with a set of such edges form a *hypergraph* (or simply a graph) if each edge is connected, by its tentacles, to its *attachment* nodes. A graph is equipped with a sequence of external nodes identified by distinct names. External nodes can be seen as the connecting points of a graph with its environment (i.e. the context). Graphs are considered in this paper up to isomorphism. For example in our case study, Figure 2a. shows a graph for an instance with a country, two airports, a plane inside each airport and one passenger in the second airport with his luggage and ticket. It has edges with labels $\widehat{country}$, $\widehat{airport}$, $\widehat{plane}$ (they represent locations), *pass*, *luggage* and *ticket*. All locations have a tentacle to a node identifying them (going up) and other tentacles for identifying their contents. $\widehat{country}$ has a tentacle to a node to connect its airports and $\widehat{airport}$ has one for planes and one for passengers. $\widehat{plane}$ has one tentacle connected to its location and one to passengers onboard (in the case of Figure 2 planes have no passengers). The passenger has a tentacle that identifies him and is connected to his location, one referencing his departure country and two for luggage and ticket. Note that a passenger with luggage and ticket form a graph and are not hierarchical (graphs are leaves in hierarchy). As you can see, the reference to the country is global with respect to location hierarchy. Also, in Figure 2b. you can see the equivalent "boxed" representation of locations for the derived graph.

We represent hypergraphs as well formed *syntactic judgments* generated from a set of axioms and inference rules. Correspondence proof can be found in [8]. Then, a graph is described as a judgment $\Gamma \vdash G$ where nodes correspond to names, with external nodes representing names in Γ (free names). Term G contains edge basic terms of the form $L(x_1, \ldots, x_n)$, where x_i are names and $L \in LE$ and operator $|$ puts together two graphs sharing their nodes. The ν operator is added to allow restriction of names. Formally defined:

Definition 3.1 [Graphs as Syntactic Judgments] Let $\mathcal{N}$ be a fixed infinite set of names and LE a ranked alphabet of labels. A *syntactic judgment* (or simply a judgment) is of the form $\Gamma \vdash G$ where $\Gamma \in \mathcal{N}^*$ is a sequence of names (the interface of the graph) and G is a term generated by the grammar:

$G ::= L(\tilde{x}) \mid G|G \mid \nu y.G \mid nil$, where $\tilde{x}$ is a vector of names, L is an edge label with $rank(L) = |\tilde{x}|$ and y is a name. Let $fn(G)$ denote the set of all free names of G, i.e. all names in G not bound by an operator for scope restriction, ν. We demand that $fn(G) \subseteq \Gamma$. Structural axioms for associativity, commutativity and identity for operation $|$ over *nil* are defined, together with axioms for alpha conversion with respect to restricted names and the usual interplay between ν, *nil* and $|$: $\nu x.\nu y.G \equiv \nu y.\nu x.G$, $\nu x.G \equiv G$ *if* $x \notin fn(G)$,

$\nu x.G \equiv \nu y.G[y/x]$ *if* $y \notin fn(G)$, $\nu x.(G_1|G_2) \equiv (\nu x.G_1)|G_2$ *if* $x \notin fn(G_2)$.

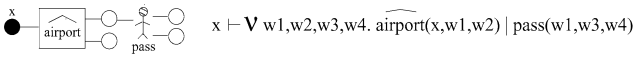
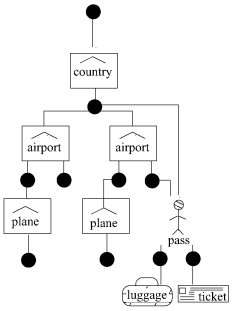
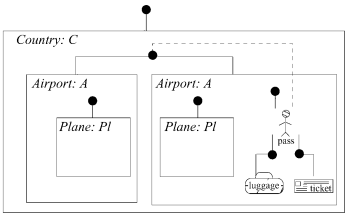


Fig. 1. Graph and Syntactic Judgment.

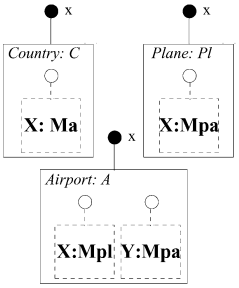
For example, Figure 1 shows a graph and a corresponding judgment with two edges and labels $\widehat{airport}$ and *pass*. Node x is visible and the rest (w_i) are restricted by ν. We write νX, with $X = \bigcup x_i$, to abbreviate $\nu x_1.\nu x_2 \ldots \nu x_n$. Using the axioms for alpha conversion and ν, for any judgment we always have an equivalent normal form $\Gamma \vdash \nu X.G$, with G a subterm containing only parallel composition of edges. Γ and X are disjoint. $\Gamma \vdash nil$ corresponds to a graph with no edges. We use notation Γ, x to denote concatenating x to Γ, assuming $x \notin \Gamma$, and Γ_1, Γ_2 as the concatenation of Γ_1 and Γ_2.



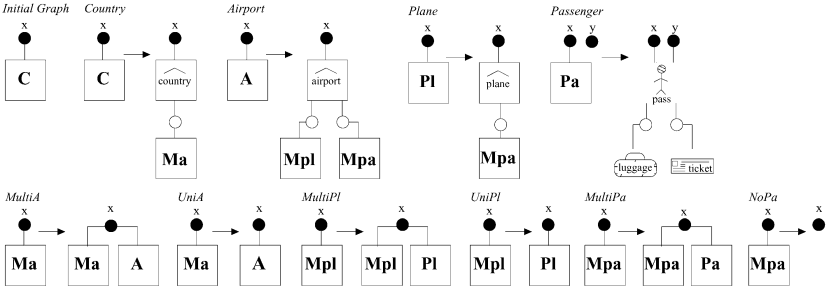
(a) Derived Graph.



(b) Hierarchical Graph



(c)Constructors



(d) Shape Rules.

Fig. 2. Airport Case Study.

3.2 Global Reference Type System

Our intention is to propose a general structure of inference rules but for space limitations, in this paper we directly present one possible instantiation that allows handling global references. This case also shows how the approach is more general than hyperedge replacement and specially relevant with respect to the integration of shapes and hierarchies. Hierarchy can be treated as a strict structure where nesting implies that elements cannot know other elements outside their same level, i.e. there are no global references. Nevertheless, many programming languages allow global references, and architectural or design languages like UML make use of global references.

First, we define shape rules and graphs as judgments. For this, we extend the notion of judgment to *type judgment* (Section 3.2.1) by adding second order variables and defining a ranked signature of basic types that type variables and judgments. Inference rules are of the form $\frac{J \quad R}{RJ}$, where J, R and RJ are type judgments with RJ the result of *replacing* J in R. Then, given a set of judgments as axioms for a shape, the derived judgments and their typing proofs define the instances of the shape. Each type system instantiation in the framework contains one inference rule and a set of axioms (Section 3.2.2). In general, we define a type system as follows:

Definition 3.2 [Type System] A type system $TS =< \mathcal{A}, \mathcal{I} >$ is defined as a set of *type judgments* for axioms $\mathcal{A}$ and an *inference rule* $\mathcal{I}$ over judgments. A *proof term* is a composition of judgment names. A proof term is well typed if a typing proof exists that generates it. Here, we consider standard proof terms without parenthesis corresponding to a unique standard proof for a produced judgment. Then, when we say proof term we mean standard proof term [3].

3.2.1 Type Judgments

Graph terms contain edges (as in Definition 3.1) and variables that work as placeholders (for other graphs) connecting some nodes (as for edges). Graphs and shape rules are type judgments. Basic types identify nonterminal symbols and variables represent nonterminal instances appearing in rules and derived graphs using them. Types and variables are ranked according to the number of nonterminal tentacles.

Definition 3.3 [Basic Types and Variables] We define a fixed set of ranked types **NT** where for $T \in NT_n$, $rank(T) = n$; and a set Var of ranked variables where $X : T \in Var$ indicates that X has type $T \in$ **NT** and $rank(X) =$

[3] Each proof term with parenthesis corresponds to a unique typing proof.

$rank(T)$. Variables are written in uppercase letter. For simplicity, we overload function *rank*. We define a special type ε used for typing graphs. Variables in Var cannot be of type ε.

In our example we need to define for the shape rules (described below) in Figure 2d. the following set of basic types: $\mathbf{NT} = \{C, A, Pl, Pa, Ma, Mpl, Mpa\}$.

Definition 3.4 [Type Judgments] Let $\mathcal{N}$ be a fixed infinite set of names, **NT** a fixed set of ranked types and Var a set of ranked variables. A type judgment is of the form $S = \Gamma, \Delta \vdash G : T_s$ where,

(i) S is a name for the judgment and $\Gamma \in \mathcal{N}^*$ is a sequence of distinct names.

(ii) Δ is a sequence of variables of the form $X_1 : T_1, \ldots, X_n : T_n$, for all variables appearing in term G. Variables in Δ are all distinct.

(iii) G is a term generated by the grammar defined in Definition 3.1 plus variables: $G ::= X(\tilde{y}) \mid L(\tilde{z}) \mid G|G \mid \nu x.G \mid nil$, with $rank(X) = rank(T_X) = |\tilde{y}|$.

(iv) $T_s \in \mathbf{NT} \cup \varepsilon$ is the resulting type of the complete judgment.

For judgments of the form $\Gamma, \Delta \vdash G : T_s$, nonterminal graphs correspond to $\Delta \neq \emptyset$ and $T_s = \varepsilon$, and terminal graphs to $\Delta = \emptyset$ and $T_s = \varepsilon$. For a shape rule ($s = L \to R$) (i.e., a graph rewriting rule) the interpretation is as follows: Type $T_s \in NT$ corresponds to the nonterminal in L, and $\Gamma, \Delta \vdash G$ is the syntactic judgment for graph R. Here Δ is the sequence of variables for the new nonterminals generated by the rule (the correspondence of rules and judgments is up to variable renaming and ordering). Sequence Γ contains the external nodes with $|\Gamma| \geq rank(T_s)$ where the *greater than* is needed for shape rules with global references (see Section 3.5 and rule *passenger* below). A rule with only terminals in R corresponds to $\Delta = \emptyset$. In this presentation variable occurrences in G are all distinct. Two edges may have the same (nonterminal) label (i.e. same type) but they are considered different instances (i.e. different variables). We use sequences for node names and variables because we need to match nodes and to identify the order of variables for the construction of proof terms. $\Delta, X : T_X$ denotes that X is the last variable of the sequence. Δ_1, Δ_2 is the concatenation of Δ_1 and Δ_2.

For example, Figure 2d. shows the shape rules and initial graph from which graphs in the shape are produced. These rules generate a flat tree structure of locations. For example, judgments for some rules and the initial graph are:

$$Init \stackrel{def}{=} x, X : C \vdash X(x) : \varepsilon \qquad Country \stackrel{def}{=} x, X : Ma \vdash \nu w.\widehat{country}(x, w) \mid X(w) : C$$
$$Airport \stackrel{def}{=} x, X : Mpl, Y : Mpa \vdash \nu w_1, w_2.\widehat{airport}(x, w_1, w_2) \mid X(w_2) \mid Y(w_1) : A$$
$$Passenger \stackrel{def}{=} y, x \vdash \nu w_1, w_2.pass(x, y, w_1, w_2) \mid luggage(w_1) \mid ticket(w_2) : Pa$$

The inclusion relation between locations defines a nested structure. Rules

Country, *Airport* and *Plane* create the hierarchical structure. *Passenger* creates a passenger instance with its luggage (for simplicity we just put one) and its ticket. The rest of the rules create a number of airports, planes and passengers [4]. Figure 2a. shows a derived graph from these rules. We do not show the derivation for space limitations. For simplicity and following the approaches in literature for shapes [6,11], wherever possible, we have presented rules in a context-free style (i.e., one nonterminal in the left hand side) with little extension to deal with global references.

3.2.2 Inference Rule

The following inference rule shows the composition of two judgments by replacing variable X in R using J obtaining as a result a new judgment RJ.

Definition 3.5 [Global References Inference Rule] Let R and J be two type judgments; then we define an inference rule of the form,

$$\frac{J = \tilde{y}, \tilde{x}, \Delta_1 \vdash H : A \qquad R = \tilde{y}, \Gamma, \Delta_2, X{:}A \vdash \nu\tilde{w}.G|X(\tilde{z}) : C}{RJ = \tilde{y}, \Gamma, \Delta_2, \Delta_1 \vdash \nu\tilde{w}.G|H\left[\tilde{z}/\tilde{x}\right] : C} \quad \begin{array}{l} |\tilde{x}| = rank(A) \\ \tilde{x} \ \cap \ (\Gamma \cup \tilde{w}) = \emptyset \end{array}$$

Judgment J represents a shape rule and it replaces in R the last variable X, which must have the same type as J (i.e., A). Judgment J is a rule with label A as left hand side producing graph $\tilde{y}, \tilde{x}, \Delta_1 \vdash H$, where $\tilde{x}$ is the sequence of attachment nodes for the rule (i.e. $rank(A) = rank(X) = |\tilde{x}|$). Judgment R corresponds to a graph or a rule (depending if C is ε or not), containing a nonterminal edge with symbol A. The resulting judgment RJ (with name concatenation RJ as proof term) is obtained by replacing in R variable X with graph H. This is done by concatenating variables Δ_2, Δ_1 from R and J (except X) and by replacing in the graph term for R the instance of X with H substituting $\tilde{x}$ with $\tilde{z}$ ($[\tilde{z}/\tilde{x}]$). You can see $\tilde{x}$ as the formal parameters and $\tilde{z}$ as the actual ones. Clearly, $|\tilde{x}| = |\tilde{z}|$ as $\tilde{z}$ represents the attachment nodes to the context where H is embedded. Nodes in $\tilde{z}$ can be both in Γ or $\tilde{w}$.

As an example we can construct a proof (without global references) corresponding to a rule for a country with an airport ($Country \ \ UniA \ Airport$).

$$\frac{UniA = x, X : A \vdash X(x) : Ma \qquad Country = z, \ Z : Ma \vdash \nu w. \ \widehat{country}(z, w) \mid Z(w) : C}{Country \ UniA = z, \ X : A \vdash \nu w. \ \widehat{country}(z, w) \mid X(w) : C}$$

$$\frac{\begin{array}{c} Airport = r, W : Mpl, Y : Mpa \vdash \nu w_1, w_2.\widehat{airport}(r, w_1, w_2) \mid W(w_1) \mid Y(w_2) : A \\ Country \ UniA = z, X : A \vdash \nu w_3.\widehat{country}(z, w_3) \mid X(w_3) : C \end{array}}{\begin{array}{c} Country \ UniA \ Airport = \\ z, W{:}Mpl, Y{:}Mpa \vdash \nu w_1, w_2, w_3. \ \widehat{country}(z, w_3)|\widehat{airport}(w_3, w_1, w_2)|W(w_1)|Y(w_2) : C \end{array}}$$

[4] We assume at least an airport and a plane per airport.

When we deal with judgments representing a hierarchical structure, the inference rule takes into account global references by adding a sequence of distinguished node names $\tilde{y}$ that are known by both rules R and J. Names in $\tilde{y}$ are not part of rank of A and are used in H as references to nodes in the external context of R, separately from the embedding of $\tilde{x}$ where in the case of a hierarchy $\tilde{z} \subseteq \tilde{w}$ corresponding to internal references of R. For example, if we take term *(Airport MultiPa)* and apply rule *Passenger* we have a rule for a passenger in an airport with a reference to the external node (r in the proof) identifying the airport country (see Figure 2).

$$\frac{\begin{array}{c} Passenger = r, x \vdash \nu w_3, w_4.pass(x, r, w_3, w_4) \mid luggage(w_3) \mid ticket(w_4) : Pa \\ Airport\ MultiPa = r, W : Mpl, X : Mpa, Y : Pa \vdash \nu w_1, w_2.\widehat{airport}(r, w_1, w_2) \mid W(w_1) \mid X(w_2) \mid Y(w_2) : A \end{array}}{\begin{array}{c} Airport\ MultiPa\ Passenger = r, W : Mpl, X : Mpa \vdash \nu w_1, w_2, w_3, w_4. \\ \widehat{airport}(r, w_1, w_2) \mid W(w_1) \mid X(w_2) \mid pass(w_2, r, w_3, w_4) \mid luggage(w_3) \mid ticket(w_4) : A \end{array}}$$

The resulting type C of RJ is the same as of R. If R is a graph with type ε, then RJ corresponds to the application of rule J over nonterminal A in R producing a new graph with type ε. If R is a rule with C as left hand side, then RJ is a composite new rule whose application is equivalent to the result of replacing R (which requires C) followed by replacing J over A (produced by R). For example, the above proofs are rules but if we join them and apply the result to the initial graph (we show only the proof term) then we produce a judgment for a nonterminal graph: *Init Country UniA Airport MultiPa Passenger*

As we mentioned, our intention is to propose different instantiations of the framework. We have presented one for global references but also it is worth mentioning that if we remove the global references from the above rule, we obtain a rule corresponding to context-free hyperedge replacement (HR) [14].

$$\frac{J = \tilde{x}, \Delta_1 \vdash H : A \qquad R = \Gamma, \Delta_2, X : A \vdash \nu\tilde{w}.G \mid X(\tilde{z}) : C}{RJ = \Gamma, \Delta_2, \Delta_1 \vdash \nu\tilde{w}.G \mid H\left[\tilde{z}/\tilde{x}\right] : C} \qquad \begin{array}{l} |\tilde{x}| = rank(A) \\ \tilde{x} \cap (\Gamma \cup \tilde{w}) = \emptyset \end{array}$$

Theorem 3.6 (Correspondence for HR Systems) *Given a HR system with a set of productions P, and a type system $TS =< \mathcal{A}, \mathcal{I} >$ with the above inference rule and as $\mathcal{A}$ the judgments for P, then there is a one-to-one correspondence of proof terms in TS and syntactic trees in HR.*

3.2.3 Hierarchical Constructors

Usually hierarchical structure is modeled by a containment relation depicted as boxes containing elements that can be boxes again. Examples of this at the level of design or architecture are subsystems or locations for mobile systems. We propose to use *hierarchy constructors* which are typed in the same way as shape rules, but initially their bodies are not defined. Namely, a hierarchical

graph is specified as a proof term containing some constructors for which only the type is given. We can define the type of a judgment as follows.

Definition 3.7 [Judgment Types] Given a type judgment of the form $J = \tilde{x}, \Delta \vdash H : A$ we can say that the type of J is given by tuple $\mathcal{T}_J = \langle \tilde{x}, \Delta, A \rangle$, consisting of the sequence $\tilde{x}$ of connecting points to the context, the shapes Δ of its arguments, and its final type A.

For example, Figure 2c. shows location constructor $Country$ with type $\mathcal{T}_{Country} = \langle (x),\ X{:}Ma,\ C \rangle$ where [5] $rank(Ma) = rank(C) = 1$. Note that we do not give a term for the right hand side of $Country$ because for hierarchy constructors we are interested in the containment relation only. What the constructor type says is that content [6] of $Country$ is X of type Ma and that its resulting type for the context is of type C with one external connection.

Anyhow, if we want to be able to check the correct typing of a term using $Country$, then we need a judgment for it to build typing proofs. A standard choice is to obtain from the proof term a *flattened* version of the hierarchical graph where a *symbolic* definition of the constructors is given. For example, we can define the constructor with an equivalent flattening rule $Country^f$ that has the same type as $Country$. In this case, the flattening rule corresponds to the one in Figure 2d. Edge label $\widehat{country}$ is a terminal symbol that identifies the root of each subtree occurrence for the constructor in a syntactic tree. Again, Figure 2c. shows the graphical representation of the location constructors for the case study and Figure 2c. the flattening rules. Then, Figure 2b. corresponds to the hierarchical graph for Figure 2a. using the constructors.

Definition 3.8 [Hierarchical Type System] Given the type system TS with axioms $\mathcal{A} = \mathcal{R} \cup \mathcal{C}$, where $\mathcal{R}$ is a set of judgments and $\mathcal{C}$ is a set of constructors, then we define the hierarchical type system TS^f where, TS^f is the same as TS but with axioms $\mathcal{A} = \mathcal{R} \cup \mathcal{C}^f$ where $\mathcal{C}^f$ is the set of flattening rules for constructors in $\mathcal{C}$. Then, a proof term represents a hierarchical graph in TS if it uses constructors from $\mathcal{C}$, and a proof in TS^f can be obtained using the flattening versions from $\mathcal{C}^f$.

In this way any proof term with hierarchical constructors can be reduced to a judgment in TS^f. The judgment for the flat version of the hierarchy is in one-to-one correspondence to the "boxed" view of the graph.

[5] We use context free examples for a simpler explanation

[6] A constructor type may contain more than one variable. This means that a box can have several contents which must be totally ordered.

3.3 Reconfiguration

After system reconfiguration it is necessary to check shape consistency. The decisions taken to obtain the final system may provide important information about the process of construction. In our framework, this corresponds to examining proof terms. To reconfigure a systems in a consistent way we propose proof term rewriting. A reconfiguration is defined as a *transformation* rule $R = (I \Rightarrow O) : T$ that takes a proof term I and returns O (both of the same type T), where O may be constructed with different rules than I. Given the modular way typing proofs are obtained, we can apply transformations over proof terms (containing the input pattern) and obtain a new proof. Note that transformations can be applied with any context and instantiation in sequence and in parallel. Reconfiguration consistency is achieved by checking that I and O have the same type: as long as cutting and pasting well typed proofs still yields well typed proofs, subject reduction is guaranteed.

A simple transformation for the example can be $MinDistance$ that changes production $MultiA$ with hierarchical constructor $MultiA50$ (to be redefined later) specifying the constraint that the minimum distance between airports should be fifty kilometers. $MinDistance = MultiA \Rightarrow MultiA50 : \langle x, X{:}Ma, Y{:}A, Ma \rangle$

4 Conclusions and Future Work

We propose a framework for representing hierarchical SA shapes using types. Our goal is to give a formal basis to model and analyze systems where hierarchy is a relevant aspect and evolution of their architectures is a common event. We have a linear representation of graphs and hierarchies suitable for analysis and a graphical representation suitable for modeling. The approach can be instantiated to obtain more expressive power allowing to choose which is the most convenient solution for some specific design problem.

For hierarchies, as future work we have to study in more detail related areas that we already mentioned, like process calculi [4] and concurrent systems [13]. Also, in the context of AGILE, we will continue our work on supporting extensions of UML for mobility [3].

We have seen in detail a type system for shapes with global references, but the study has to continue on other possible instantiations. With respect to shape analysis, we consider that some of the results in the related work for pointer data structures can be relevant to our approach as well, specially given the correspondence with Hyperedge Replacement stated in Theorem 3.6.

Finally, it is worth mentioning that we believe possible to give a translation of our type systems into λ-calculus: Type judgments are mapped to type judgments in the λ-calculus with very similar term structure. The application

of the inference rule corresponds to abstraction of the variable over which the rule is applied together with application and β-reduction. Thus it should be possible to obtain a convenient implementation in a functional language of suitable tools for supporting our approach.

References

[1] Andrade, L. and et al. AGILE: Software architecture for mobility. In *Recent Trends in Algebraic Develeopment Techniques—16th International Workshop, WADT 2002, Frauenchiemsee*, volume 2755 of *LNCS*. Springer-Verlag, 2003.

[2] Bakewell, A., Plump, D., and Runciman, C. Specifying pointer structures by graph reduction. Tech. Rep. YCS-2003-367, Dep. of CS, Univ. of York, 2003.

[3] Baumeister, H., Koch, N., Kosiuczenko, P., and Wirsing, M. Extending activity diagrams to model mobile systems. In *Intl. Conf. NetObjectDays, 2002. Revised Papers*, volume 2591 of *LNCS*, pages 278–293. SV, 2003.

[4] Cardelli, L. and Gordon, A. Mobile ambients. In Maurice Nivat, editor, *FoSSaCS'98*, volume 1378 of *LNCS*, pages 140–155. Springer-Verlag, 1998.

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

[6] Fradet, P. and Le Métayer, D. Shape types. In *Principles of Programming Languages (POPL'97)*, pages 27–39. ACM Press, 1997.

[7] Garlan, D. and Shaw, M. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[8] Hirsch, D. *Graph Transformation Models for Software Architecture Styles*. PhD thesis, Dept. of Computer Science, Universidad de Buenos Aires, May 2003.

[9] Hirsch, D. and Montanari, U. Higher-order hyperedge replacement systems and their transformations: Specifying software architecture reconfigurations. In *GRATRA 2000*, Tech. Rep., TU Berlin, 2000-02, pages 215–223, 2000.

[10] Hoffmann, B. Abstraction and control for shapely nested graph transformation. *Fundamenta Informaticae*, No. to appear, 2003.

[11] Le Métayer, D. Describing software architecture styles using graph grammars. *IEEE Transactions on Software Engineering*, 24(7), July 1998.

[12] Medvidovic, N. and Taylor, R. A classification and comparison framework for software architecture description languages. *IEEE TSE*, 26(1):70–93, 2000.

[13] Milner, R. Bigraphical reactive systems. In *Concur '01*, volume 2154 of *Lecture Notes in Computer Science*, pages 16–35. Springer-Verlag, 2001.

[14] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation: Foundations*, volume 1. World Scientific, 1997.

[15] Wermelinger, M. and Fiadeiro, J. A graph transformation approach to software architecture reconfiguration. *Science of Computer Programming*, 44:133–155, 2002.