Consistent Transformations for Software Architecture Styles of Distributed Systems

Dan Hirsch [a,1] and Ugo Montanari [b,2]

[a] *Departamento de Computación, Universidad de Buenos Aires, Ciudad Universitaria, Pab.I, (1428), Buenos Aires, Argentina*
dhirsch@dc.uba.ar

[b] *Dipartimento di Informatica, Universita' di Pisa, Corso Italia 40, (56125), Pisa, Italia*
ugo@di.unipi.it

Abstract

One major problem for the specification and verification of software architectures and specially with distributed systems, is when system evolution includes dynamic changes and reconfigurations of components and connections. This paper presents a method for specifying reconfigurations or *transformations* over the topology of the architecture style, being sure that if the transformation can be specified, then its application over the system will be consistent with respect to the expected architecture style configuration. Styles are described by context-free hyperedge graph grammars. In this context, an instance of an architecture style is determined by a graph generated by the grammar.

The formalization of the method will be introduced in two ways. The first approach is a visual presentation and has the intention of showing that it can be used by software architects in real life in a simple way. For this, as we mentioned, graph grammars represent styles and their generated graphs specific system architectures. A transformation is visualized as a rule that takes a graph generated by the grammar and applies a reconfiguration to part of that graph, obtaining a new graph that is surely another valid graph of the grammar. The second approach presents a formal model for the first one using lambda-terms and tile sequents showing also that it is implementable.

[1] Research supported in part by ARTES Project, PIC 11-00000-01856, ANPCyT and TW72, UBACyT.

[2] Research supported in part by CNR Project *Metodi per Sistemi Connessi mediante Reti*; by MURST project *Tecniche Formali per Sistemi Software*; by TMR Network GETGRATS; and by Esprit Working Groups *APPLIGRAPH* and *COORDINA*.

1 Introduction

In previous work we presented a formalism for modeling software architecture styles using graph grammars [7]. One major problem for the specification and verification of software architectures and specially with distributed systems, is when system evolution includes dynamic changes and reconfigurations of components and connections. In this line of work we present here a method for specifying reconfigurations (called *transformations* from now on) over the topology of the architecture style, being sure that if the transformation can be specified, then its application over the system will be consistent with respect to the expected architecture style configuration.

A simple and natural way to describe a system architecture is by using graphs, and as an extension of this, by using grammars to describe styles. So a grammar will generate all possible system instances of that style. In our work we represent a system as a graph where edges (or hyperedges) are components and nodes are ports of communication. In the previous paper we also used constraint solving to model system evolution and controlled dynamics, but here for a simpler presentation of the work we do not include constrained rules. The productions here are used to construct configurations of systems and transformations are applied over the topology of the architectures.

The idea of reconfiguration consistency is not new. In [9], Daniel Le Metayer presented a method for specifying reconfiguration rules and for verifying that the result is consistent with the initial architecture. The difference between the two approaches is that in [9] styles are specified as context free graph grammars, but reconfigurations are specified outside the grammar by context sensitive rules. Then it has to be verified that the style structure is maintained, i.e. the result of the reconfiguration is consistent with the grammar. In our approach, instead, reconfiguration rules automatically preserve the style.

In our work, styles are described by context-free graph grammars, i.e. productions are applied over individual edges. So, a grammar consists of an initial graph and a set of rules that generates the language of the grammar.

In this context, an instance of an architecture style is determined by a graph generated by the grammar. The application of a transformation obtains from a graph, after some reconfigurations, another graph. For each graph there is a set of proofs of the grammar that are the possible ways of constructing it from grammar rules. A transformation is applied over a proof segment and returns a new proof segment. Then, the idea is that to apply a transformation over a graph you have to find a proof of that graph that matches the transformation. After applying the transformation, its result corresponds to the reconfigured segment of a graph proof. This result is part of a valid proof of the graph obtained by the desired reconfiguration. This assures that a transformation is between valid proofs. It is important to mention, that because transformations are over a proof segment they can be composed and applied to several parts of a graph. Also, after a transformation is obtained, you can start from any of the proofs of the new graph allowing for other transformations to be applied.

The formalization of the method will be introduced in two ways. The first approach is a visual presentation and has the intention of showing that it can be used by software architects in real life in a simple way. For this, as we mentioned above, graph grammars represent styles and their generated graphs specific system architectures. A transformation is visualized as a rule that takes a graph generated by the grammar and applies a reconfiguration to part of that graph, obtaining a new graph that is surely another valid graph of the grammar.

The second approach presents a formal model for the first one and at the same time shows that it is implementable. The productions of the graph grammars are codified as second order functions of a simply typed lambda calculus language. A codification of a proof is a term over the language. Then, a transformation codification takes a term that represents a proof segment and returns another term. This is obtained using a *tile sequent* [5,3].

A tile logic consists of a sequent calculus over a rule-based model (i.e., the *tile model*), whose rules define the behavior of certain *open* (e.g., partially specified) *configurations*, which may interact through their *interfaces*. Then, the behavior of a system as a whole consists of a coordinated evolution of its sub-systems.

A tile sequent has the form, $s \xrightarrow[b]{a} s'$, stating that the *initial configuration* s of the system evolves to the *final configuration* s' producing the *effect* b, which can be observed by the rest of the system. However, such a step is allowed only if the subcomponents of s (which is in general an open configuration) evolve to the subcomponents of s', producing the *trigger* a.

Triggers and effects are called *observations*. Tiles can be composed horizontally (synchronizing an effect with a trigger), vertically (computational evolutions of a fixed component), and in parallel (concurrent steps) to generate larger steps. Thanks to these composition properties, there exists a precise correspondence between tiles and the cells of suitable *double categories*.

The tile sequent gives us in an automatic way the certainty that the transformation is done between valid proofs. The tile sequent is applied over a proof segment and it can be applied (if possible) over different proof segments simultaneously allowing for composition of transformations.

In the present case, for the visual presentation, interfaces are tuples of nonterminals, configurations are proof segments and vertical arrows express the correspondence between nonterminals before and after the transformation. For the codification, interfaces correspond to types of the terms of proof segments.

It is clear that our aim is to give architects a tool to specify in a consistent way complex reconfigurations over the architectures they are working with. This is fundamental for software architecture modeling because once a transformation is obtained and its correctness checked (that starts and ends with valid proofs), it can be included in a library of transformations for its future use. Also, it is important to note that correctness and applicability of a transformation can be automated, and that codification of rules in their functional version is direct and implementable.

In section 2 the basic notions of hyperedge graphs, graph rewriting, and simply typed λ-calculus are introduced. Software architectures and the tile model are introduced in sections 3 and 4, respectively, and in section 5 an application example is used to describe the proposed method. Section 6 presents our conclusions and future work.

2 Background

2.1 Graphs and Graph Rewriting

Definition 2.1 (Hypergraphs) *We define an edge-labeled hypergraph, or simply a graph as a tuple $G = \langle N, E, c, lab_{LE} \rangle$, where:*

(i) *N is a set of nodes.*

(ii) *E is a set of edges.*

(iii) *$c : E \mapsto N^*$ is the connection function (each edge can be connected to a list of nodes).*

(iv) *$lab_{LE} : E \mapsto LE$ is the labeling function of edges, where the alphabet LE of edge labels is ranked, i.e. $LE = \{LE_n\}_{n=0,1,\ldots}$, where LE_n contains the n-hyperedge labels.*

A *graph production* rewrites a graph into another graph, deleting some elements (nodes and edges), generating new ones, and preserving others. In this paper we will just consider context-free productions, which rewrite a single n-hyperedge, into an arbitrary graph, while preserving the nodes connected by the rewritten hyperedge. Therefore, in a context-free production, no nodes are deleted.

Definition 2.2 (Graph Productions) *A graph production p is a pair $\langle L, R \rangle$ where:*

(i) *L is a n-hyperedge label.*

(ii) *R is a graph.*

(iii) *$f : 1, .., n \mapsto nodes(R)$.*

Context-free graph productions will be written as $L \to R$, where L is the label of the n-hyperedge to be rewritten and R is the graph to be generated. A production $p = (L \to R)$ can be applied to a graph G yielding H ($G \Rightarrow_p H$) if there is an *occurrence* of L in G. The result of applying p to G is a graph H which is obtained from G by removing the occurrence of L and embedding a fresh copy of R in G by coalescing the corresponding nodes.

Definition 2.3 (Graph Rewriting System) *A graph rewriting system is a pair $GRS = \langle G_0, P \rangle$, where:*

(i) *G_0 is a graph.*

(ii) *P is a set of graph productions.*

A derivation for GRS is a finite sequence of direct derivation steps of the form $G_0 \Rightarrow_{p1} G_1 \Rightarrow_{p2} \ldots \Rightarrow_{pn} G_n = H$, where $p_1, \ldots, p_n$ are in P.

2.2 λ-calculus

To fix the notation, we recall the classical simply typed λ-calculus. In particular, we consider typed λ-terms in $\lambda^{\to,\times,\epsilon}$, i.e., with *product types* and *unit*. We refer to [15] for an extensive presentation of the subject.

A *λ-signature* Σ is a pair $\langle B, C \rangle$, where B is the set of *type constants* (i.e., the basic types allowed), and C is the set of (typed) *term constants*. The *types* are built over the type constants $b \in B$ using the following grammar:

$$\sigma ::= b \mid \epsilon \mid \sigma \times \sigma \mid \sigma \to \sigma$$

Since term constants must be typed, the set C is usually represented as a collection $\{c_1 : \sigma_1, ..., c_n : \sigma_n, ...\}$, where σ_i is the type of the term constant c_i, for $i = 1, ..., n, ...$

A *term* (also *λ-term*) M over Σ with variables $x \in X$ can then be constructed accordingly to the syntax below. The set of all λ-terms over signature Σ is called λ_Σ.

$$M ::= c \mid \star \mid x \mid \lambda x : \sigma.M \mid MM \mid \langle M, M \rangle \mid Proj_1^{\sigma,\tau} M \mid Proj_2^{\sigma,\tau} M$$

However, we are only interested in well-typed terms. Since terms can contain variables, the *type judgments* must be parameterized with respect to suitable *assignments* of types to the (free) variables. A generic assignment Γ has the form $\{x_1 : \sigma_1, ..., x_k : \sigma_k\}$. Since we will consider cartesian closed categories with chosen products as suitable models, we assume that each assignment is a list, instead of a set, and that each variable can appear at most once in the list. We use the notation $\Gamma, x : \sigma$ to denote the assignment obtained by appending $x : \sigma$ at the end of Γ. Similarly, we will write also Γ_1, Γ_2 to state that the assignment is the concatenation of Γ_1 followed by Γ_2.

A type judgment has the form

$$\Gamma \vdash M : \sigma,$$

which means that the term M is well-typed with type σ under the assumption

$$\frac{c : \sigma \in C}{\emptyset \vdash c : \sigma} \qquad \frac{}{\emptyset \vdash \star : \epsilon} \qquad \frac{}{x : \sigma \vdash x : \sigma}$$

$$\frac{\Gamma \vdash M : \sigma}{\Gamma, x : \tau \vdash M : \sigma} \qquad \frac{\Gamma_1, x : \tau, y : \rho, \Gamma_2 \vdash M : \sigma}{\Gamma_1, y : \rho, x : \tau, \Gamma_2 \vdash M : \sigma}$$

$$\frac{\Gamma \vdash M : \sigma \quad \Gamma \vdash N : \tau}{\Gamma \vdash \langle M, N \rangle : \sigma \times \tau}$$

$$\frac{\Gamma \vdash M : \sigma \times \tau}{\Gamma \vdash Proj_1^{\sigma,\tau} M : \sigma} \qquad \frac{\Gamma \vdash M : \sigma \times \tau}{\Gamma \vdash Proj_2^{\sigma,\tau} M : \tau}$$

$$\frac{\Gamma, x : \tau \vdash M : \sigma}{\Gamma \vdash \lambda x : \tau . M : \tau \to \sigma} \qquad \frac{\Gamma \vdash M : \tau \to \sigma \quad \Gamma \vdash N : \tau}{\Gamma \vdash MN : \sigma}$$

Table 1
Well-typed judgments

of Γ. The correct type judgments for terms over the signature $\langle B, C \rangle$ are those generated by applying the rules in Table 1.

Notice that each type judgment $x_1 : \sigma_1, ..., x_n : \sigma_n \vdash M : \sigma$ or equivalently a closed term $\lambda x_1 : \sigma_1, ..., \lambda x_n : \sigma_n.M : (\sigma_1 \times \ldots \times \sigma_n) \to \sigma$ is represented as an arrow from type $\sigma_1 \times \cdots \times \sigma_n$ to σ. We omit the axiomatization that identifies equivalent terms and the corresponding rules for reduction to normal form. Also a lambda theory can have proper axioms in addition to the general axioms.

The operations of identity and composition over closed terms id_τ, $(M; N)$ and $(M \otimes N)$ are defined as:

Identity:

$$id_\tau \;=\; \lambda x : \tau.\ x \;: \tau \to \tau$$

Sequential Composition:

$$\frac{M : \tau \to \sigma\ ,\ N : \sigma \to \rho}{M; N \;=\; \lambda x : \tau.\ N(Mx) \;: \tau \to \rho}$$

Parallel Composition:

$$\frac{M : \tau \to \sigma\ ,\ N : \rho \to \pi}{M \otimes N \;=\; \lambda < x, y >: \tau \times \rho. < Mx,\ Ny > : (\tau \times \rho) \to (\sigma \times \pi)}$$

3 Software Architecture

Due to the increment in size and complexity of software systems, the problem of software design is no more just a choice of algorithms and data structures, requiring a more abstract analysis of the design. When a system is built based on several components, the system design and general structure organization appears as a new type of problem. As system complexity increased it is fundamental for all the people involved in the development process to possess a notion of its general structure. But this creates a new problem: the choice of the right abstraction level for system modeling. The analysis of a system general structure includes topics like: global control structures, and coordination; functionality assignment to a design element; choice among design alternatives. This level of design is called software architecture [14].

A formal characterization of the software architecture of a system allows us expose the architectural structure of the system. A software architecture style is a class of architectures exhibiting a common pattern. This allows to abstract details of particular components giving a way to categorize paradigms of architectures and to obtain new instances of systems from a specific style. For the description of software architectures many methods have been used. Recently, research for producing architecture description languages (ADLs) included revision of semantic models used in other areas. We can mention work done with languages and models as Z [1], CHAM [8], Darwin [11], Wright [2]. Other projects for the development of ADLs are ACME [4], Rapide [10] and more [13].

One major problem in the specification and verification of an architecture and specially with distributed systems, is when system evolution includes dynamic changes of components and connections, reconfiguration and mobility [12]. Some of the existing approaches for modeling software architectures styles deal with these problems but only partially. Others only allow for fixed description of styles, components and connectors.

The aim of our work is to propose a general framework that allows specification and analysis of software architectures and software architecture styles. Also it is of special interest to attack in an efficient way the problem of reconfiguration and dynamic changes mentioned above. A first approach has been already presented in [6,7] for the specification of software architecture styles.

A simple and natural way to describe a system architecture is by using graphs, and as an extension of this, by using grammars to describe styles. The work presented in [6,7] represents a system as a graph where edges (or hyperedges) are components and nodes are ports of communication. In this paper we use this notation to describe styles as context-free productions of a graph grammar, as introduced in section 2.1. Productions are applied over individual edges. So, a grammar consists of an initial graph and a set of context-free rules. In this context, an instance of an architecture style is determined by a graph produced by the grammar.

4 The Tile Model

We now describe the basic features of the tile model. The presentation of the tile model follows [5,3].

A tile logic consists of a sequent calculus over a rule-based model (i.e., the *tile model*), whose rules define the behavior of certain *open* (e.g., partially specified) *configurations*, which may interact through their *interfaces*. Then, the behavior of a system as a whole consists of a coordinated evolution of its sub-systems. The name "tile" is due to the two-dimensional graphical representation of such rules, which have the form in Figure 1 but can also be written as sequent,

$$g \xrightarrow[b]{a} h$$

stating that the *initial configuration* g of the system evolves to the *final configuration* h producing the *effect* b, which can be observed by the rest of the system. However, such a step is allowed only if the subcomponents of g (which is in general an open configuration) evolve to the subcomponents of h, producing the *trigger* a.

In this version configurations and effects are lambda terms on different signatures, sharing the type constants. The signatures are different since the horizontal term constants essentially represent hyperedge labels, while vertical term constants (when present: in the example of section 5 there are no vertical term constants) represent actions needed for modeling constraints.

Definition 4.1 (tile sequent, tile rewrite system) *Let Σ_H and Σ_V be two λ-signatures, called the horizontal and the vertical signature, respectively, sharing the same type constants B.*

A $\Sigma_H - \Sigma_V$ tile sequent is a quadruple $g \xrightarrow[b]{a} h$, where $g : u \to v$ and $h : w \to z$ are closed terms on Σ_H (up to equivalence), while $a : u \to w$ and $b : v \to z$ are closed terms on Σ_V. Terms g, h, a and b are called the initial configuration*, the* final configuration*, the* trigger*, and the* effect *of the tile, respectively. Trigger and effect are called observations. Types u, v, w and z are called the* initial input interface*, the* initial output interface*, the* final input interface*, and the* final output interface*, respectively.*

A tile rewrite system (trs) $\mathcal{R}$ is a triple $\langle \Sigma_H, \Sigma_V, R \rangle$, where R *is a set of $\Sigma_H - \Sigma_V$ sequents called rewrite rules.*

A TRS $\mathcal{R}$ can be considered as a logical theory, and new sequents can be derived from it via certain inference rules.

Definition 4.2 (tile logic) *Let $\mathcal{R} = \langle \Sigma_H, \Sigma_V, R \rangle$ be a* TRS*. Then we say that* R *entails the class* **R** *of the tile sequents $g \xrightarrow[b]{a} h$ obtained by finitely many*

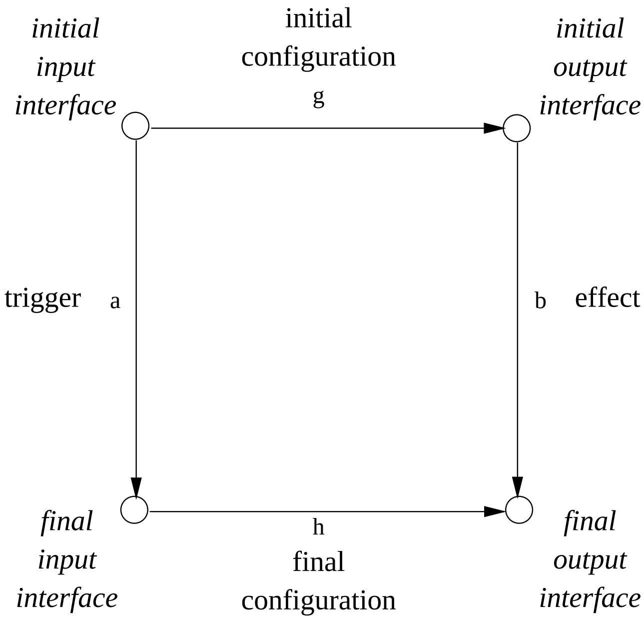

Fig. 1. A tile

applications of the following inference rules

basic rules:

$$(generators)\frac{g \xrightarrow[b]{a} h \in R}{g \xrightarrow[b]{a} h \in \mathbf{R}}$$

$$(h-refl)\frac{g : u \to w \in \lambda_{\Sigma_H}}{g \xrightarrow[id_w]{id_u} g \in \mathbf{R}} \qquad (v-refl)\frac{a : u \to w \in \lambda_{\Sigma_V}}{id_u \xrightarrow[a]{a} id_w \in \mathbf{R}};$$

composition rules:

$$(p-comp)\frac{g \xrightarrow[b]{a} h, g' \xrightarrow[b']{a'} h' \in \mathbf{R}}{g \otimes g' \xrightarrow[b \otimes b']{a \otimes a'} h \otimes h' \in \mathbf{R}}$$

$$(h-comp)\frac{g \xrightarrow[c]{a} h, g' \xrightarrow[b]{c} h' \in \mathbf{R}}{g; g' \xrightarrow[b]{a} h; h' \in \mathbf{R}}$$

$$(v-comp)\frac{g \xrightarrow[b]{a} k, k \xrightarrow[b']{a'} h \in \mathbf{R}}{g \xrightarrow[b;b']{a;a'} h \in \mathbf{R}};$$

auxiliary tiles:

Auxiliary tiles are defined as all the sequents,

$$g \xrightarrow[b]{a} h$$

with no term constants (i.e. those in Σ_H and Σ_V), where the closed terms $g; b$ and $a; h$ are equivalent.

Basic rules provide the sequents corresponding to transformations, together with suitable identity tiles. Composition rules provide all possible ways in which sequents can be composed. Tiles can be composed horizontally (synchronizing an effect with a trigger), vertically (computational evolutions of a fixed component), and in parallel (concurrent steps) to generate larger steps. Thanks to these composition properties, there exists a precise correspondence between tiles and the cells of suitable *double categories*. Auxiliary rules provide all the consistent rearrangements of the interfaces.

5 Application Example

Now we will apply the notions introduced in the previous sections to an example. The structure used is well known and will let us introduce the proposed method.

The idea of the proposed method is to give software architects a formal language for specifying, at the software architecture level of description, the reconfigurations and dynamic changes that a system can suffer during its evolution. In this case we do not introduce a new language but complement the one used to describe styles (i.e. graph grammars). This allows to add the specification of transformations to already specified styles without the need of specifying them for a second time in a new specific language for reconfigurations.

The use of this method is intended in two forms (assuming that graph grammars are used to describe styles). The first one is at the time when a designer wants to specify a reconfiguration for systems of a specific style. The second one is when a designer wants to apply one or more transformations, that are already specified and included in a library, to the style of the system he is working with. In the first case, to specify a transformation, the designer has to find a proof for the subgraph where he wants the reconfiguration to happen, and a proof for the resulting subgraph after the reconfiguration takes place. With these proofs and using the operations that we present in the next sections, the desired transformation is formally specified and can be added to the style description, being sure that it is consistent with respect to the style. For the second case, if a designer wants to apply an already specified transformation to follow the dynamic evolution of a specific system instance, it means that he has its graph specification, so he can apply the transformation to a proof that matches the initial proof segment of the transformation and obtain the new proof for the reconfigured system.

The example is presented in two parts. The first part is a visual presentation using graphs and graph grammars to specify systems and rules between proofs for their transformations. The second part is the corresponding codification using λ-terms for grammar rules and proofs and tile sequents for transformations.

5.1 Graph View

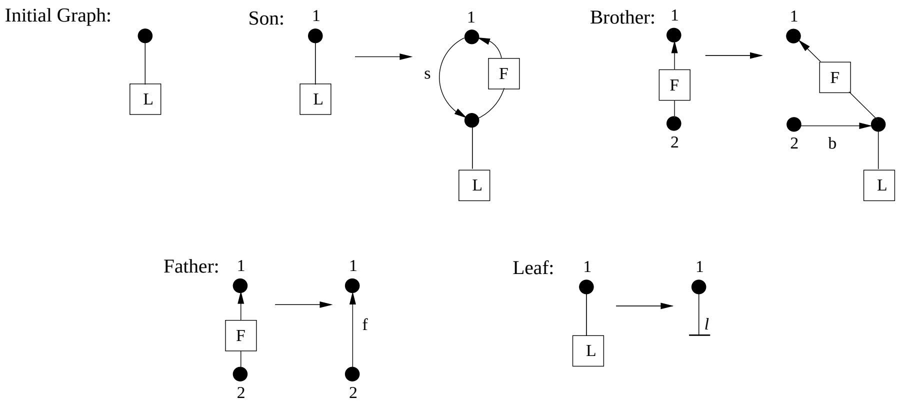

Fig. 2. Graph Grammar

Systems produced by the grammar in figure 2 have a tree-like structure. The grammar is composed of four rules. Rule *Son* produces, from a leaf (**L**) edge, a subtree creating a new node connected by a son (**s**) and a father (**F**) edges. Rule *Brother* adds a brother (**b**) edge at the end of a line of brothers. A line of brothers is always connected to its parent by a son and a father edges at the beginning and the end of the line, respectively. In this example only two non-terminals are needed, **F** and **L**. *Father* and *Leaf* are their terminal rules. Figure 3 shows an example of derivation applying rules *Son* and *Brother* that it is not closed.

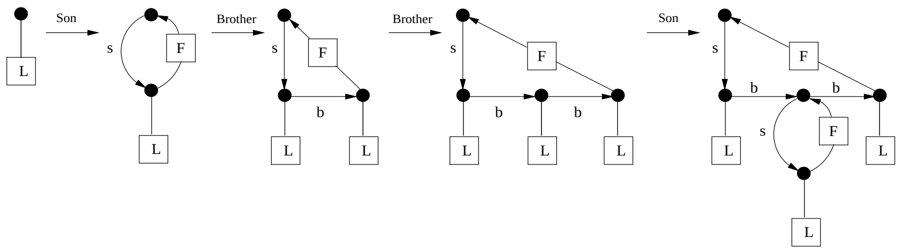

Fig. 3. A derivation

Now, we want to specify a reconfiguration of a tree where any two subtrees that are neighbors (their roots are in the same line of brothers), can be permuted. This is achieved by composition of two smaller transformations. The first transformation permutes the first two subtrees of a line of brothers and the second transformation permutes any other pair of adjacent subtrees (i.e. their roots are connected by a brother edge).

As mentioned in the introduction, transformations are applied over a proof segment of the grammar giving as result another proof segment. So, we have to find the correct initial proof for the transformations being specified. For the first transformation to permute the first two subtrees, the initial proof can be obtained by applying rules *Son* and *Brother*, as in the proof in figure 3. Actually, the initial proof takes the form of a *derived rule* (see figure figure 4a) which can be obtained by sequentially composing rules *Son* and *Brother*.

Figure 4a) is the rule in the standard derivation notation and figure 4b) is the same rule as it is used in the transformation. In this notation tuples of non terminals are in the extremes and represent the interfaces to the rest of the proof. In the left extreme, you have the non-terminal hyperedge label from the left side of the rule in figure 4a) and at the right extreme, the non terminals on the right side of the rule. The horizontal composition (i.e. matching the nonterminals and removing them) of productions applied in a derivation will generate the corresponding graph.

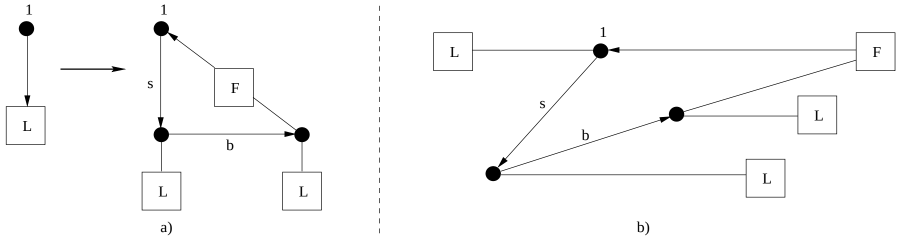

Fig. 4. Initial rule for permutation of the first two subtrees

After specifying the initial rule, we have to find what will be the final rule of the transformation and how the desired transformation between them will be specified. In this case it is easy to find the final rule, it is the same as the initial one. This is because the initial rule can generate any two subtrees, so the new proof is the same as the initial one with the corresponding derivations permuted, but at this level of the derivation it makes no difference on the proof. These is not enough to describe the transformation because as we said the rule in figure 4 is general enough to generate any subtrees. We need to specify the relation from the initial to the final rules, and this is done by using a set of operations over the graph proofs that allows modeling reconfigurations and how they are applied.

Figure 5 shows a three dimensional representation of the transformation to permute the first pair of subtrees. Dashed lines represent the vertical axis. The north and south parts of the figure are the initial and final rules of the transformation. As we mentioned above, tuples of non terminals are the interfaces to the rest of the proof. Vertical arrows express the correspondence between non terminals before and after the transformation. In this case, the final rule remains the same as the initial one except that the two subtrees given by the non-terminal leaves are permuted as can be seen in the figure.

To apply this transformation to a system graph and obtain the new reconfigured system we need an adequate proof. Given a proof of a system graph that matches the initial proof segment of the transformation, the graph proof is divided in three sections. A first section from the initial graph of the gram-

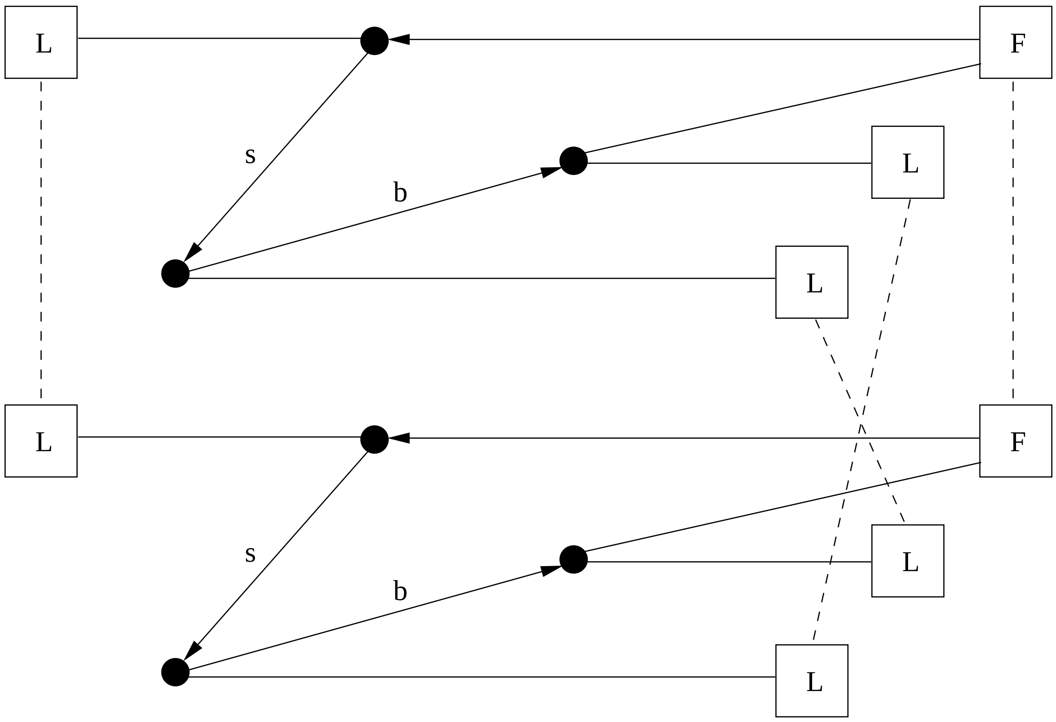

Fig. 5. Transformation for permutation of the first two subtrees

mar to the proof segment that corresponds to the transformation initial rule. A second section that matches the transformation initial proof segment. And a third section for the rest of the proof. Then, the reconfigured graph is given by the proof constructed by the composition of the identities of the first and third segment of the original proof with the proof segment resulting from the application of the transformation. It is obvious that these compositions to be completed must satisfy the interfaces of the different sections of the proofs.

The second transformation for permuting two adjacent subtrees is shown in figure 6. This transformation applied together with the first one, achieved the permutation of any two subtrees where their roots are in the same line of brothers.

Certain auxiliary transformations corresponding to the auxiliary tiles of definition 4.2 are also needed. They are used (if needed) for rearrangements of interfaces to allow the recomposition of proofs after a transformation is applied. Composition of the transformations in figure 5 or figure 6 with the one in figure 7 allows to rearrange the interfaces to connect the rest of the proof. In this example the vertical signature Σ_V is empty, but transformations which employ constraints might require the horizontal composition of several rewrite rules. In fact, triggers and effects, which are lambda terms on Σ_V, play the role of synchronization requirements on the rules to be composed.

The method lets other transformations to be applied in parallel to different

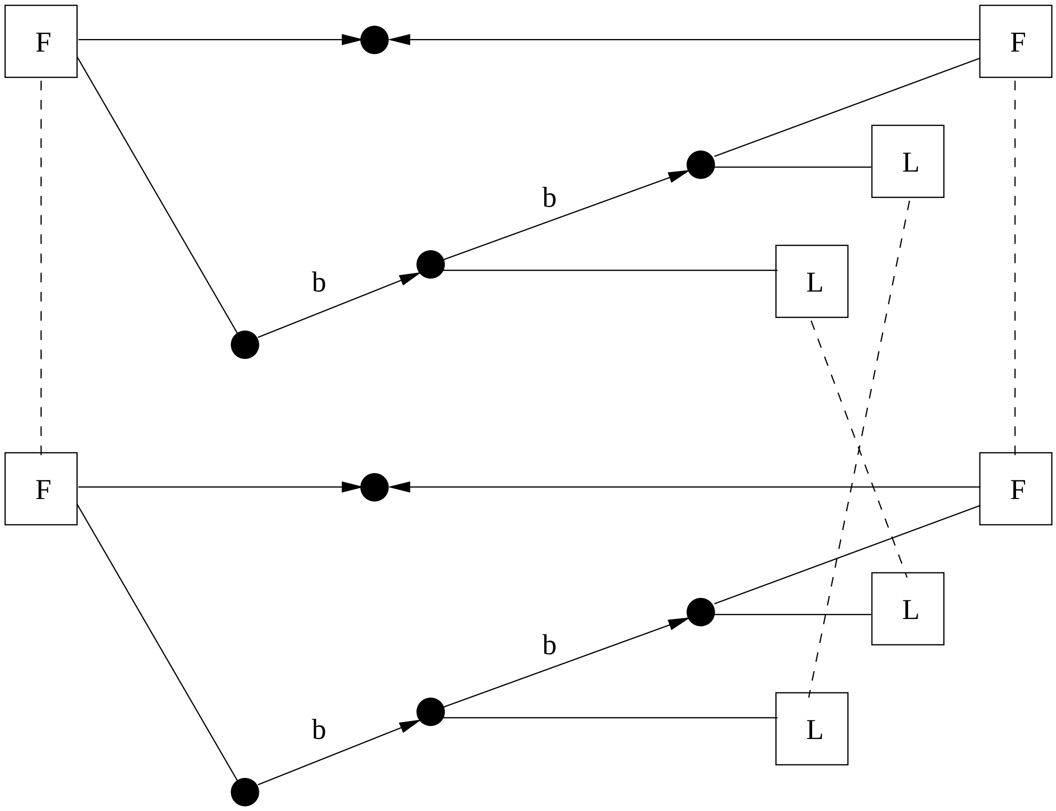

Fig. 6. Transformation for permutation of any two adjacent subtrees

proof segments. In this way it is possible to specify independent reconfigurations that a system can suffer simultaneously. Also, given that a graph determines a set of possible proofs of the grammar, then after a transformation is applied and a reconfigured system is obtained, a new transformation can be applied over the new system just using the corresponding proof for it. And this is independent from the proof used in the first transformation.

5.2 Presentation of the example using λ-terms

In this section we present a codification of the graph view of the proposed method. In this way, we obtain a semantics for the grammar rules, graph proofs and transformations and at the same time show that the method is implementable. It is worth mentioning that the codification from the graph view is direct and can be automated. Grammar rules and graph proofs are codified in second-order lambda terms and transformations are tile sequents with second-order lambda terms for the horizontal and vertical signatures. Then, the operations that are used to rearrange proofs correspond to the operations of sequential and parallel composition ($\_\,;\_$ and $\_ \otimes \_$) presented in section 2.2 and applied to lambda terms on Σ_H. As example, we codified the grammar rules from the previous section and the transformation to permute

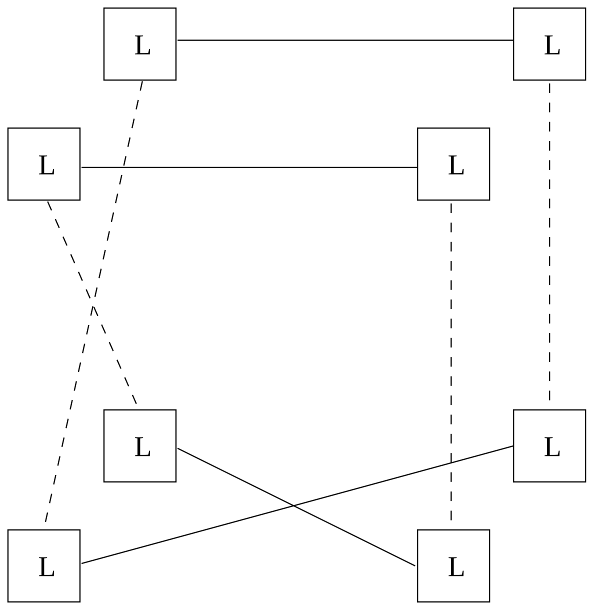

Fig. 7. Coordination Transformation

the first two subtrees of a line of brothers.

First we codified the grammar rules. For this we use second-order lambda terms of a simply typed lambda calculus defined below. As a convention for denoting typed variables we use n, m for variables of type 1, p, q, r for variables of type $*$ and x, y, z for variables of type $(k \to *)$. Terms (of any type) are denoted by M, N.

λ-signature:

$$\Sigma = \langle C, B \rangle$$

Type Constants:

$$C = \{1\ ,\ *\}$$

Term Constants:

$$B = \{b,\ s,\ f : 2 \to *\ ,\ l : 1 \to *\ ,\ | : * \times * \to *\ ,\ \nu : (1 \to *) \to *\}$$

Axioms:

$$p \mid q = q \mid p \qquad\qquad \nu m.\nu n.M = \nu n.\nu m.M$$

$$p \mid (q \mid r) = (p \mid q) \mid r \qquad M \mid \nu n.N = \nu n.M \mid N\ ,\ n \notin fv(M)$$

Edges are described as arrows. Terms $b,\ s,\ f,\ l$ define the terminal edges of the grammar. The arity of an arrow is given by the number of nodes the edge is attached to. The $|$ operator is used to put together edges. The ν operator is for the new nodes that the rules create and term $\nu(\lambda n.M)$ is abbreviated as $\nu n.M$. It is important to distinguish between constants of the method (i.e. $|$ and ν) and constants of this particular example (i.e. b, s, f, l). In this example there are two non terminals, edges **L**, **F**. If it is needed, extra type constants can be introduced. For this case, the two only non terminals have different number of attaching points, so there is no need for this.

Now we present the definition of the terms for the rules of figure 2. The type of a term is determined by the non terminals that appear in the rule.

Rules:

$$Son \ : \ (1 \rightarrow *) \times (2 \rightarrow *) \rightarrow (1 \rightarrow *)$$

$$Son \stackrel{def}{=} \lambda(x_L, x_F).\ \lambda n.\ \nu m.\ s(n,m) \mid x_L\ m \mid x_F(m,n)$$

$$Brother \ : \ (1 \rightarrow *) \times (2 \rightarrow *) \rightarrow (2 \rightarrow *)$$

$$Brother \stackrel{def}{=} \lambda(x_L, x_F).\lambda(n_1, n_2).\ \nu m.\ b(n_1,m) \mid x_L\ m \mid x_F(m,n_2)$$

$$Leaf \stackrel{def}{=} \lambda n.\ l\ n \ : \ 1 \rightarrow *$$

$$Father \stackrel{def}{=} \lambda(n_1, n_2).\ f(n_1, n_2) \ : \ 2 \rightarrow *$$

Initial Graph:

$$Init \stackrel{def}{=} \lambda x_L.\ \nu m.\ x_L\ m \ : \ (1 \rightarrow *) \rightarrow *$$

Note that, as non terminals define the interfaces of transformations in the graph view, here types will do the same with tile sequents. Then, types define how proofs terms are constructed and the sequents composed.

Now, we define the term for the initial rule in figure 4 and then the sequent for the transformation in figure 5.

Initial Rule:

$$\lambda(x_l, y_L, x_F).\ \lambda n.\ \nu m.\ \nu m'.\ s(n,m) \mid x_L\ m \mid b(m,m') \mid y_L\ m' \mid x_F(m',n)$$
$$: (1 \rightarrow *) \times (1 \rightarrow *)\ \times (2 \rightarrow *) \rightarrow (1 \rightarrow *)$$

Also it can be written as,

$$x_L,\ y_L,\ x_F \vdash$$
$$\lambda n.\ \nu m.\ \nu m'.\ s(n,m) \mid x_L\ m \mid\ b(m,m') \mid y_L\ m' \mid x_F(m',n) : 1 \rightarrow *$$

Transformation:

$$x_L,\ y_L,\ x_F \vdash$$
$$[\lambda n.\ \nu m.\ \nu m'.\ s(n,m) \mid x_L\ m \mid\ b(m,m') \mid y_L\ m' \mid x_F(m',n)\ /\ z]\ \ z$$
$$\Longrightarrow$$
$$[y_L/x_L,\ x_L/y_L,\ x_F/x_F]\,(\lambda n.\ \nu m.\ \nu m'.\ \ s(n,m) \mid x_L\ m$$
$$\mid\ b(m,m') \mid y_L\ m' \mid x_F(m',n)) : 1 \rightarrow *$$

To understand the above notation, it should first be noticed that the type judgment style is here used for representing tile sequents. Namely, given a tile sequent

$$\lambda x : \pi.H : \pi \to \tau \quad \xrightarrow[\lambda y:\tau.U:\tau\to\sigma]{\lambda x:\pi.V:\pi\to\rho} \quad \lambda z : \rho.G : \rho \to \sigma$$

it is represented as,

$$x : \pi \vdash [H/y : \tau]U \Rightarrow [V/z : \rho]G : \sigma$$

where the formal substitutions in square parenthesis are prefixed to the terms they apply to rather than postfixed, according to the diagrammatic style of [15]. Substitutions $[H/y : \tau]$ and $[V/z : \rho]$ are formal rather than actual, since their role is not to be applied to the terms they refer to, but rather to define the north and west sides of the tile sequent and at the same time to provide assignments for the (non closed) terms U and G, respectively. In addition, it should be observed that in the visual presentation of figure 5 (respectively 6) the "arguments" **L**, **L** and **F** are on the right, while the "result" **L** (respectively **F**) is on the left. This is consistent with the ordinary top-to-bottom and left-to-right representation of graph derivations. However this visual presentation is *inverted* with respect to the ordinary direction of lambda terms, where the variable types are on the left and result types are on the right. Thus to get a faithful correspondence between the visual presentation and its codification, one needs to flip the former left-to-right. For instance, in this way the east side (U) of the tile turns out to be an identity, as it is the case (z) for the above codification.

It is possible to prove by induction on the rules that the tiles entailed by the rewrite system consisting of figures 5 and 6 represent exactly the transformations which permute in all possible ways the subtrees in the same lines of brothers, for lines at all depths.

6 Conclusions and Future Work

This work has introduced a method that allows the specification of general reconfiguration rules for software architecture styles of distributed systems.

The advantage of this approach is that we present a visual language for describing such transformations and at the same time we provide a formal basis for it and show that it is implementable by using lambda terms and tiles sequents. This method permits the inclusion of specifications of dynamic changes and reconfigurations of systems in style descriptions. Although, libraries of transformations for specific styles can be created, from where architects can choose which transformations their systems will be allowed to. Another point to comment is, (taking grammars as the style description language) that we did not introduce a new language for describing transformations besides the one for software architecture styles, but complement it with a new facility.

Also notice that transformation rules can be composed to create more complex transformations and that they can be applied simultaneously over different parts of a system architecture. This is a good characteristic of the method because independent reconfigurations can be modeled and consecutive reconfiguration steps can be followed.

As future work, we want to continue research on topics of reconfigurations and dynamic changes of systems. This method offers a good starting point for experimenting with more complex systems. The idea of formally describing complex transformations in a general way, can help in specifying reconfiguration patterns usually found during the evolution of distributed systems.

References

[1] Allen, R., Garlan, D.: A Formal Approach to Software Architectures. Proc. IFIP Congress, September 1992.

[2] Allen, R., Garlan, D.: Formalizing Architectural Connection. Proc. International Conference on Software Engineering, May 1994.

[3] Bruni, R. and Montanari, U., Cartesian Closed Double Categories, their Lambda-Notation, and the Pi-Calculus, invited talk, Proc. 14th Symposium on Logic in Computer Science, Trento, July 2-5, 1999, IEEE Computer Society, 1999.

[4] Garlan, D., Monroe, R., Wile, D.: Acme: An Architecture Description Interchange Language. Proceedings of CASCON '97, November 1997.

[5] Gadducci, F., Montanari, U.: The Tile Model. In: Gordon Plotkin, Colin Stirling, and Mads Tofte, Eds., Proof, Language and Interaction: Essays in Honour of Robin Milner, MIT Press, to appear. Paper available from http://www.di.unipi.it/ ugo/festschrift.ps.

[6] Hirsch, D., Inverardi, P., Montanari, U.: Graph Grammars and Constraint Solving for Software Architecture Styles. Proceedings of the Third International Software Architecture Workshop, Orlando, E.E.U.U., November 1-2, 1998.

[7] Hirsch, D., Inverardi, P., Montanari, U.: Modeling Software Architectures and Styles with Graph Grammars and Constraint Solving. Solving. In Proceedings of the First Working IFIP Conference on Software Architecture, San Antonio, Texas, E.E.U.U., February 22-24, 1999.

[8] Inverardi, P., Wolf, A., Yankelevich, D.: Checking Assumptions in Component Dynamics at the Architectural Level.Submitted for Publication, 1997.

[9] Le M'etayer, D. : Describing Software Architecture Styles using Graph Grammars. IEEE Transactions on Software Engineering . Vol. 24, Nr 7, July 1998.

[10] Lukham D., Kenney, J., Augustin L., Vera, J., Bryan D., Mann W.: Specification and Analysis of System Architecture Using Rapide. IEEE Transactions on Software Engineering, Vol. 21, Nr 4, 1995.

[11] Magee, J., Kramer, J.: Dynamic structure in software architectures. Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering, ACM Software Engineering Notes, 1996.

[12] Magee, J., Kramer, J.: Self organising software architectures. Proceedings of the Second International Software Architecture Workshop, 1996.

[13] Medvidovic, N.: A classification and comparison framework for software architecture description languages. Technical Report ICS-TR-97- 02, University of California, Irvine, Department of Information and Computer Science, 1997.

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

[15] Mitchell, J.: Foundations for Programming Languages. MIT Press, 1996.