

Style-Based Reconfigurations of Software Architectures with QoS Constraints

R. Bruni¹, A. Lluch Lafuente², U. Montanari³, E. Tuosto⁴

^{1,2,3}Department of Computer Science, University of Pisa

⁴Department of Computer Science, University of Leicester

¹bruni, ²lafuente, ³ugo@di.unipi.it ⁴et52@mcs.le.ac.uk

Abstract. We present *Architectural Design Rewriting* (ADR), a graph-based approach to deal with the design of reconfigurable software architectures. The key features we promote are: (i) hierarchical design; (ii) soft constraints for modeling QoS attributes; (iii) style-preserving reconfigurations; (iv) rule-based approach; and (v) algebraic presentation. Roughly, actual architectures are modeled by graphs with port and attribute nodes through which component edges are connected. Uniformly, QoS constraints on attributes are also modeled as edges. Architectures are designed hierarchically by a set of edge replacement rules that fix the architectural style. Depending on their reading, productions allow: (i) top-down design by refinement, (ii) bottom-up typing, and (iii) algebraic composition of typed architectures, with terms corresponding to style proofs. Moreover, productions exploit constraint primitives that can be used in the refinement phase, e.g., to reserve a certain amount of resources or to postpone architectural decisions. Similarly, reconfigurations are modeled as graph transformations triggered by constraints primitives to guarantee that certain levels of QoS are maintained. The main contribution is showing that we can take advantage by exploiting styles when specifying reconfigurations: (i) by giving hierarchical specifications that exploit the classes introduced by the style, (ii) by guaranteeing that all reconfigurations are style-preserving, (iii) by expressing reconfigurations as ordinary term rewrite rules on the algebra of style proofs. Overall, this results in a simple and formal mechanism for designing architectures according to a style, for checking that an architecture is an instance of a style and for ensuring style preservation during reconfigurations.

1 Introduction

The architecture of a software system basically consists of the structure of components and the way they are interconnected. When designing an architecture, it is desirable to consider the concept of *architectural style* [23], i.e., some set of rules indicating which components can be part of the architecture and how they can be legally interconnected. Typical architectural styles include client-server and pipelines. Architectural styles can be applied to reuse existing design patterns. In addition, they offer a further benefit when architectural information is carried over the execution of the system, since one can control whether changes

style must be preserved. Thus, reconfigurations model how the old configuration is transformed into a new one, preferably preserving the architectural style. The way this is done is via a set of reconfiguration rules that express the necessary changes in the architecture. The rules of our approach are style preserving and are defined at any abstraction level of the architecture.

Contribution. The main contribution of our approach is a simple algebraic formalism called *Architectural Design Rewriting* (ADR). It can be used for describing architectural development issues like refinement, abstraction and composition, and run-time issues like execution and reconfiguration. Additional contributions of our approach are the inclusion of QoS in architectures by means of constraints, and thus in architectural styles, design and reconfiguration, and the definition of style-based reconfiguration rules at any abstraction level of the architecture. Our proposal can be seen as a formal basis for extending existing architectural description languages [20] (ADLs) with the novel concepts of our approach. In addition, we believe that our approach contributes to emphasize the benefits of considering architectural aspects in system design, especially to ensure quality measures of the implemented system [13].

We emphasize here the algebraic reading of both style-based design and reconfiguration rules. Design rules (or productions) can be seen as the basic operations for composing well-defined architectures according to their types, yielding well-typed results. Hence, architectural information can be kept at run-time just by recording the proof term composed by functional application of the productions used in the design phase. Style-preserving reconfigurations can then be expressed as ordinary term rewriting rules on such architectural information. The hierarchical nature of the approach allows for the specification of rules that take typed architectures as parameters. Pushing further this view, one sees that sometimes rules can be conveniently applied in both directions, and their employment is instrumental to the effective application of directional rewrite rules. This scenario is well known in the area of term rewriting, where rewriting up-to equational axioms has received lot of attention.

While term rewrites can always occur in any larger context, an additional issue might be the use of conditional reconfiguration rules, where transformations can be derived using several inference rules expressing that a composed architecture can be rewritten only provided that its sub-components are suitably transformed first. This step would make the formalism very powerful. On the one side ordinary process algebras like CCS or π -calculus should become immediately representable, yielding a unified setting where design development, runtime execution and reconfiguration could be defined on the same foot. On the other side complex reconfigurations, e.g., achieving nested wrappings of repetitive subcomponents could become easily expressible and checkable.

Related Work. The use of graphs and graphs transformations to model architectural styles has been proposed by several authors (see [23], for instance) which based on the concept of *shapes* in programming languages [11]. Amongst such works our paper is closely related to the approach of [17] which we extend with

QoS constraints and simpler, more intuitive and powerful formalism. Within this research line, our algebraic approach is original and reconfigurations triggered by constraints are supported here for the first time.

The paper is also related to approaches that deal with reconfigurations in software architectures defined by an ADL. As far as we are aware, our approach differs from such approaches in that they do not consider QoS attributes as constraints or hierarchical reconfigurations. In [1], for instance, QoS aspects are modeled at the architectural level in order to specify *QoS contracts* among components that are then monitored during run-time: QoS contracts can trigger run-time reconfigurations of the system, however they are not used for constraining hierarchical reconfigurations of the system design.

Another difference is that we use graph rewritings as a unifying model to model architectural design, behavior and reconfiguration, while most ADLs use different formalism for such issues. For instance, various approaches [14] mix the ingredients of a concrete ADL and the Alloy [18, 19] language to respectively describe the vocabulary and constraints that define an architectural style.

Running example. Along with the paper we illustrate our approach with an architectural style that basically considers two classes of concrete components: *agents* and *gateways*. Agents have two ports that are used to interconnect them with other agents. A sequence of connected agents is abstracted as a *chain*. Maximal chains must be cyclic thus forming *rings*. All the rings of the *system* must be attached via gateway to the same point in a star-fashion. The abstract and concrete components that we have introduced have QoS attributes associated. Rings have a certain capacity and a workload that cannot exceed the capacity. The workload of a ring is the workload of its maximal chain. The workload of a chain is the sum of the workloads of its component agents.

Figure 1 shows a simple instance of the (informally) described style with two rings (dotted boxes) respectively formed by two and one agents (*a*-labeled boxes). Rings are connected to the center of the star (empty circle x_c) via gateways (*g*-labeled boxes). The workload of the rings is represented by the filled circles w_1 and w_4 , respectively, and constraints represented by the Σ -labeled boxes require the workload of a ring to be equal to the sum of the workloads of its component agents. For instance, in the left ring, w_1 is constrained to be the sum of w_2 and w_3 . Additional constraints (\leq -labeled boxes) force the workload of each ring to not exceed its capacity (c_1 and c_2 for the left and right rings, respectively).

We shall give design rules for constructing systems according to such style. The rules can be seen as refining abstract components (e.g., a system) into more concrete ones (e.g., a ring) up to arrive to the most concrete level (e.g., the agents and gateways). Then, we exemplify our approach to hierarchical style-based reconfiguration by giving rules for dealing with under and over-loaded rings. The rules are not directly specified at the most concrete level (i.e., as a manipulation of agents) as it is usual done, but a more abstract level (chains of agents, as we shall see) and are guaranteed to preserve the style.

Structure of the Paper The paper is structured as follows. Section 2 defines our notion of software architectures with QoS attributes as *designs*, which are basically graphs with interface and constraints. Section 3 explains how to define architectural styles by a set of design productions. Section 4 presents the main contribution of our approach, namely, hierarchical style-based reconfigurations by means of term rewriting. Section 5 concludes the paper and outlines future work.

2 Software Architectures

An architecture is an abstraction or view of the implementation of a system. The basic view describes the present components and their interconnections but other aspects (e.g., behavior) can also be considered, leading to an aspect oriented development. In this paper we focus on structural and QoS aspects. This section explains our formalism for representing and manipulating software architectures with such key aspects.

2.1 Architectural Models

Software architectures can be suitably represented by graphs. Take, for instance, the typical run-time component and connector view of software architectures [8] which is at the base of architectural description languages (ADLs) like ACME [12]. Basically, an architecture is viewed as a collection of interconnected components. Components model the main computational entities of the system. Their interface is represented by ports where connectors are attached. Connectors model the pathways of communication. Their attachment points are usually called roles. A simple way to see such a model as a graph is to represent components and connectors with hyperedges whose attaching points, called tentacles, represent ports and roles. A port is connected to a role when the corresponding tentacles are attached to the same node. So let us now give a formal definition for graphs.

Definition 1. A graph is tuple $G = \langle V, E, t \rangle$ where V is the set of nodes, E is the set of edges and $t : E \rightarrow V^*$ is the tentacle function.



Fig. 2. A system is as a network of rings.

Example 1. In our running example components are represented by edges, attributes by nodes to which component edges are attached. For the sake of simplicity connectors are represented by nodes, such that the interconnections of

two components is represented by the set of set of nodes they are attached to. Note that the tentacle of an edge are represented by outgoing arrows, except the first tentacle that is represented by an incoming arrow. The already described Figure 1 depicts a concrete architecture of the system. Figure 2 instead depicts the architecture of a system at the most abstract levels. The graph on the left represents the (closed) system which is an isolated component without attributes and is thus represented by an S labeled edge without tentacles. The graph on the right represents a network (N -labeled edge) attached to a node x_1 . We shall see that the figure represents a design rule that can be interpreted as refining a system as a network or, dually, composing a system from a network.

2.2 Architectural Elements

The vocabulary of an architectural style consists of a set of architectural elements. For instance, in a client-server architecture one has clearly two classes of components: clients and servers. In our running example, instead, we have agents, rings, chains, gateways, etc. A suitable way to represent architectural elements in a graphical representation of software architectures consists of using type graphs, which have one edge and node for each different type of edge and node element, respectively, that one wants to consider. One can also have a type hierarchy with sub- and super-types relations as in [3] but we restrict to a one-level typing hierarchy for the sake of simplicity. The relationship between actual elements of an architecture and the abstract classes of elements is suitably represented by graph morphisms.

Definition 2. *Given two graphs G, H a graph morphism is a pair of functions $\langle f_V, f_E \rangle$ preserving the tentacle function, i.e., $f_V \circ t_G = t_H \circ f_E$, where $f_V : V_G \rightarrow V_H$ and $f_E : E_G \rightarrow E_H$.*

Example 2. We do not depict the type graph of our running example since it results in a cumbersome figure. Instead we give its textual representation. The set of nodes is $\{\circ, \bullet\}$, where \circ is a purely structural node representing connections and \bullet nodes represent QoS attributes. The classes of components of our system are *System, Network, Ring, Chain, agent, gateway*. In addition, we have constraints \leq and Σ . These elements are represented by the set of edges $\{S, N, R, C, a, g, \leq, \Sigma\}$. The tentacle function models which type of nodes a component can be attached to and is defined by $\{S \mapsto (), N \mapsto (\circ), R \mapsto (\circ, \bullet, \bullet), C \mapsto (\circ, \circ, \bullet), a \mapsto (\circ, \circ, \bullet), g \mapsto (\circ, \circ), \leq \mapsto (\bullet, \bullet), \Sigma \mapsto (\bullet, \bullet, \bullet)\}$. In our graphical representation, types are explicitly represented, where the type of an edge is the label inside the corresponding box.

Typed graphs are hence defined as graphs equipped with a typing morphism.

Definition 3. *Let T be a graph. A typed graph G over T is a graph $|G|$, together with a graph morphism $\tau_G : |G| \rightarrow T$. A morphism between T -typed graphs $f : G_1 \rightarrow G_2$ is a graph morphism $f : |G_1| \rightarrow |G_2|$ consistent with the typing, i.e., such that $\tau_{G_1} = \tau_{G_2} \circ f$.*

Now, in our approach we distinguish two kind of architectural components: abstract (or non-terminal, refinable) and concrete (or terminal, basic, non refinable). For instance, we shall see that a chain of agents can be refined as a concatenation of chains of agents (cf. Figure 6) or as an agent (cf. Figure 5). Technically, edges E_T of the type graph T (and thus edge types) are partitioned into several alphabets. We first notice that the tentacle function applies to edges of T as well, and thus E_T is partitioned into families indexed by tuples of node types. Even if there is only one node type, edge types are ranked according to the number of tentacles. An additional distinction is between terminals \mathcal{T} and non-terminals \mathcal{NT} . As for ordinary string grammars, (non) terminal edges, namely edges labeled by a (non) terminal symbol, represent abstract components which cannot (can) be refined, or instantiated. Thus, terminal edges represent basic components of the architecture. In particular, a family of terminal symbols of special interests for our approach denotes constraints. For the sake of a clear presentation, we shall use upper-case letters to denote abstract components and any other kind of symbols for basic components.



Fig. 3. Two networks form a network.

Example 3. In our approach, for instance, *System*, *Network*, *Ring* and *Chain* are abstract classes and *agent*, *gateway* are basic classes. Indeed a network can be refined as two networks attached to the same node as illustrated in Figure 3 or as a more complex structure combining a ring, a gateway and a constraint as depicted in Figure 4.

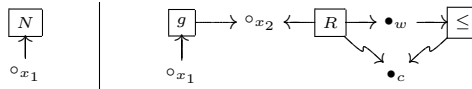


Fig. 4. A ring is a network.

In our approach a *design* is an assembly of interconnected basic components that have a typed interface represented by a non-terminal edge. The idea is that the type of the interface edge represents the abstract component class while its tentacles represent the exposed nodes.

Definition 4. A design is a graph with edge interface, i.e, a triple $d = \langle L_d, R_d, i_d \rangle$, where L_d is a (typed) graph consisting only of a nonterminal labeled by A_d and

by distinct nodes attached to its tentacles; R_d is a (typed) graph without nonterminal edges; and $i_d : V_{L_d} \rightarrow V_{R_d}$ is an injective function.

The notion of graph morphism is trivially extended to morphisms between designs. The nodes of R_d in the image of i_d are the interface nodes of the design, ordered according to the tentacles of the nonterminal edge in L_d , while the nonterminal A_d labeling the latter gives the type (or role, meaning, etc.) of the design. We shall see that this notion of design is suited for system development by refinement, abstraction or composition. Abusing of notation we shall use the term *graph* for both a typed graph with interface and its underlying graph. In addition we sometimes say that a graph is of type $A(x_1, \dots, x_n)$ (or just A) meaning that the interface of the graph is an edge of type A with nodes x_1, \dots, x_n .



Fig. 5. An agent is a chain.

Example 4. Figure 5 illustrates a simple design. The underlying graph R_d (right part of the figure) consists of an agent represented by an a -labeled edge attached to nodes x_1 and x_2 and the attribute node w . The interface L_d is a C -labelled edge representing the fact that an agent can be seen as a chain of agents. Observe that, for the sake of simplicity, we assume that the interface morphism i_d maps nodes from the interface to the nodes of the underlying graph that have the same name.

2.3 Software Architectures with QoS

We introduce QoS aspects in software architecture by means of constraints. Roughly, attributes are represented by nodes constraint edge impose a constraint on its attachment nodes. Instead of limiting the approach to *crisp* constraints that either accepted or rejected, we shall consider *soft* constraints where each possible tuple of values is assigned a certain value expressing its level of preference. Our precise flavor of soft constraints is based on constraint semirings [4] (c-semirings for short), a special kind of semirings with additional properties that make them suitable to represent constraints. We first recall the definition of semiring.

Definition 5. A semiring is a tuple $\langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$ such that A is a set; $\mathbf{0}$ and $\mathbf{1}$ are elements of A ; $+$ is commutative, associative and has $\mathbf{0}$ as unit element; \times is associative, distributes over $+$ and has $\mathbf{1}$ as unit and $\mathbf{0}$ as absorbing elements.

A *c-semiring* is a semiring $S = \langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$ such that $+$ is idempotent, $\mathbf{1}$ is its absorbing element, and \times is commutative. One can show [4] that c-semirings benefit from various properties. First, one can define a partial order \leq_S on A such that $a \leq_S b$ whenever $a+b = b$. As a result, $\langle A, \leq_S \rangle$ is a complete lattice where $+$ coincides with its least upper bound and $\mathbf{0}, \mathbf{1}$ are the bottom and top elements of the lattice.

Intuitively, A is the domain of partially ordered preference levels. This is particularly interesting when considering multiple criteria, an issue that c-semiring can suitably model since Cartesian products, power domains and functional constructions of c-semirings are again c-semirings. The additive operation $+$ is thus used to choose the the least upper bound of a set of values. The multiplicative operation is used to formalizes the combination of constraints. Finally, $\mathbf{0}$ and $\mathbf{1}$ represent the worst and best levels.

Classical (crisp) constraints are given by the Boolean semiring which is defined as $\langle \{true, false\}, \vee, \wedge, false, true \rangle$. Fuzzy constraints, instead are modeled by using the fuzzy semiring $\langle [0, 1], max, min, 0, 1 \rangle$ which has preference levels between 0 and 1. Finally, the tropical semiring $\langle \mathbb{R}^+, min, +, 0, +\infty \rangle$ which has been traditionally used to model cost notions in optimization problems can be used to represent constraints that have an associated cost.

For the sake of simplicity we assume that all the attributes have the same domain D . Then given a set V of variables over D and a semiring $S = \langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$ one can define a constraint system as a semiring $S_C = \langle \mathcal{C}, \oplus, \otimes, \hat{\mathbf{0}}, \hat{\mathbf{1}} \rangle$ whose domain $\mathcal{C} = (V \rightarrow D) \rightarrow A$ is the set of all possible S -valued constraints on variables of V with domain D [5].

The idea is that constraints are defined as functions that, given an assignment $\rho : V \rightarrow D$ of values to variables, return elements of the original semiring S . Constraints have a finite support denoted by $supp$ which is a finite subset of variables on which they actually depend. Formally, $supp(c) = \{v \in V \mid \exists \rho, a, b. c\rho[v \mapsto a] \neq c\rho[v \mapsto b]\}$. Summing and combining constraints is modeled by functions \oplus and \otimes respectively defined by $(c_1 \oplus c_2)\rho = c_1\rho + c_2\rho$ and $(c_1 \otimes c_2)\rho = c_1\rho \times c_2\rho$, while a constraint (with empty support) associating the semiring value a to each assignment is denoted by \hat{a} .

The combined constraint of a set C of constraints is defined by $\hat{\otimes} C$ and the restriction $(x)c$ of a constraint c w.r.t. a variable $x \in V$ is defined by $(x)c\rho = \sum_{a \in D} c\rho[a/x]$. Intuitively, x is eliminated by taking the least upper bound of all the values for x . Thus, the best level of consistency of a set of constraints C is $(supp(x)) \hat{\otimes} C$, i.e., the combined constraint of C restricted w.r.t. its support. If the result is $\hat{\mathbf{0}}$ we say that C is *inconsistent*.

Example 5. In our example, the only attributes are workloads and capacities both modeled by the non-negative real numbers (we have $D = \mathbb{R}^+$) and we shall consider the Boolean semiring which coincides with a Boolean algebra and yields crisp constraints. Thus, the constraints of our example basically consist of comparisons of natural numbers. However, the generality of constraint semirings is still exploited to represent constraints as values of a functional semiring.

A semiring S_C can suitably define a soft constraint system since one can define a relation $\vdash: 2^{\mathcal{C}} \times \mathcal{C}$ as $C \vdash c$ whenever $\bigoplus C \leq_{S_C} c$. Relation \vdash can be used to establish whether a constraint c is entailed by a set of constraints C . This will be useful to define conditional reconfigurations in the spirit of the *ask* operation of concurrent constraint programming [22, 5], i.e., reconfigurations that can be applied only if a condition expressed by a set of constraints is entailed by the store of constraints of the architecture where the reconfiguration is suppose to be applied.

Observe that our graphs come equipped with constraints which are part of the non-terminal edges of the graph. A design, therefore, has an associated constraint system given by the set of constraints of its underlying graph, which we shall sometimes refer to as the *store*). Moreover, a design production implicitly introduces constraints much like a *tell* operation [22, 5] on a constraint system.

Example 6. Consider again Figure 1. The topmost edges are mapped into constraints $w_1 \leq c_1$ and $w_2 \leq c_2$. On the other hand, the only Σ -labeled edge is mapped into constraint Σ that maps each triple (x, y, z) satisfying $x = y + z$ to true and each other tuple to false. In all cases, the support coincides with the attached nodes.

3 Design

Designing a software architecture is a process that might consist of putting together existing designs (composition), describing the internal structure of an abstract component (refinement) or specifying that an assembly of components are actually an abstract component (abstraction). When considering styles, all these operations should be governed by a mechanism that gives guarantees about the style of the system and its components at the different levels of abstraction. We believe that graph transformations are suitable formalisms to achieve this (see e.g [15] for an early work proposing graph transformations as architectural design formalism).

3.1 Architectural Styles

An architectural style consists of the set of components that can be part of the architecture (the vocabulary), and a set of rules indicating how they can be legally interconnected. In our approach the vocabulary of a style is given by the type graph, while the legal interconnections are given by means of sets of productions, which are very much like a design where the underlying graph can also have non-terminal edes.

Definition 6. A (design) production p is a tuple $\langle L_p, R_p, i_p, l \rangle$ where L_p is a (typed) graph consisting only of a nonterminal labeled by A_p and by distinct nodes attached to its tentacles; R_p is a (typed) graph with both terminal and non-terminal edges; $i_p : V_{L_p} \rightarrow V_{R_p}$ is an injective function; and l is a bijective function mapping the non-terminal edges of R_p on an initial segment $[1, 2, \dots, n_p]$ of positive numbers.

The *type* of a production p is $A_1 A_2 \dots A_n \rightarrow A_p$, where A_k is the nonterminal symbol labeling the k -th nonterminal edge e_k of R_p , namely with $l(e_k) = k$. The functional type $A_1 A_2 \dots A_n \rightarrow A_p$ associated to p is not an accident. In fact, p can be considered a function that when applied to a tuple $\langle d_1, d_2, \dots, d_{n_p} \rangle$ of designs of types A_1, A_2, \dots, A_{n_p} , respectively, returns a design $d = p(d_1, d_2, \dots, d_{n_p})$ of type A_p . The definition is obvious: $d = (L_p, R_d, i_p)$, where R_d is obtained from R_p by replacing edge e_k in it with graph R_{d_k} respecting the tentacle function i_{d_k} , $k = 1, \dots, n_p$. Dually, a design production can be seen as refinement of an abstract component of type A as an assembly of concrete and abstract components, the latter being of type A_1, A_2, \dots, A_{n_p} .

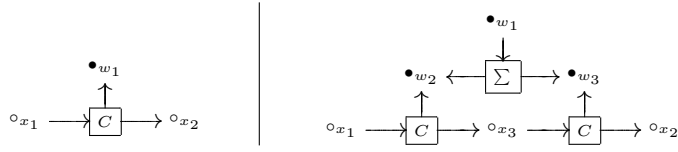


Fig. 6. Two chains form a chain.

Example 7. Consider, for instance, Figure 6. It illustrates a production **chains** that takes two designs d_1, d_2 of type C and returns a design of type C . For instance, if d_1 and d_2 were both composed by a single agent the design $\mathbf{chain}(d_1, d_2)$ would be much like the graph on the right of Figure 6 where the C labeled edges would be labeled by a . The figure can also be interpreted as a refinement rule stating that a chain can be decomposed into two consecutive chains of agents.

The construction above yields a many-sorted algebra \mathcal{M} , where sorts are nonterminal symbols \mathcal{NT} , values are designs and operations are productions of the corresponding types. Thus, a set of productions \mathcal{P} determines a family of architectural styles, where software architectures of type A are those in the carrier \mathcal{M}_A corresponding to sort A . Furthermore, if a term t of type A of \mathcal{M} evaluates to a design d , $t = d$ itself can be considered the typing proof of d .

Once we have established our algebraic setting, we can import several useful concepts from the standard algebraic machinery. A straightforward construction introduces free typed variables X into terms. As usual, an assignment η of variables to values can be uniquely extended to an evaluation $t[\eta]$ of terms with variables. Furthermore, term substitution $t[t'/X]$ can be defined. The latter construction has an obvious meaning: given a partial design $t(X)$ with a component X to be refined, and a refinement term t' (which can be considered a derived operation) of the right type, the (partial) design $t[t'/X]$ is the result of the refinement step. Conversely, given $t[t'/X]$, the identification of a subsystem t' of $t[t'/X]$, such that t' and $t(X)$ are terms in \mathcal{M} , is an abstraction step. Different levels of complexity of the construction depend on terms being linear (one occurrence of each variable) or not.

Our algebraic approach is very general. For instance, the design process can be seen as the process of writing a term for a design. A refinement design process can be seen as a top-down writing of a term, while a compositional design process can be seen as bottom-up writing of a term. Second, style checking, i.e., checking whether an architecture meets a certain style reduces to writing a term. Finally, it is worth noticing that most process algebras operations, when adapted to graphs, can be represented by simple productions. For instance, parallel composition and name restriction are easily represented by graph productions.

3.2 Design rules for the running example

We have already introduced most of the design productions of our running example. We summarize here all of them and introduce the remaining ones. In this section we follow the compositional point of view, where productions are seen as functions composing a set of typed designs into an abstract one.

The design production `sys` for composing systems is depicted in Figure 2. It forms a system from a network attached to a node.

Figure 3 depicts production `nets` which composes a net from two nets attached to the same node.

Production `net` (cf. Figure 4) produces a network from a ring, which has two attributes and is attached to the center of the star via a gateway. Attributes w and c represent the workload and capacity of the ring, respectively, and a constraint edge requires the workload of a ring to not exceed its capacity.



Fig. 7. A ring is a chain.

A ring with attributes w and c attached to a node x_1 forms as a chain in design production `ring` (cf. Figure 7). The extremes of the chain are connected to x_1 and have an attribute node w representing the workload of the ring.

The production `chains` depicted in Figure 6 represents the concatenation of two chains as a chain whose workload must be equal to the sum of the workloads of the original chains.

The last production `agent` (cf. Figure 5) constructs a chain from an agent.

4 Reconfiguration

Architectures are not static but might evolve in different dimensions. We have seen that architectures can be designed at static-time by refining abstract components or assembling subsystem architectures. During run-time instead, architectures might evolve due to actions of normal behavior or reconfigurations.

Monitoring the execution of a system in order to guarantee important aspects like performance and reliability is a fundamental task in many applications like, for instance, service oriented applications, where critical aspects such as business or security issues are involved. Carrying architectural information at run-time through the system deployment can help in this task, because critical changes in the system are directly reflected in its architecture. Components leaving or joining the system and attributes changing their values can require correcting actions that lead the system into a proper state. Such actions are called *reconfigurations*, and are specially common in systems with dynamic architectures.

For instance, in our example, rings can become over- or under-loaded. A proper reconfiguration to deal with an overloaded ring could be to create a new ring with a part of the overloaded ring in such a manner that neither the old nor the new ring are overloaded. Conversely, when two rings are underloaded another reconfiguration could merge both rings into a single one.

While reconfigurations can be naively described in terms of changes in the concrete architecture, we argue that they arise more naturally and in a well-disciplined way at an abstract level of the architecture. Indeed, the reconfigurations informally mentioned above have been written in terms of rings (abstract class level), rather than agents (basic element level). An additional issue that one would like to have in a reconfiguration mechanism is the capacity to give guarantees about the architectural style. For instance, whether it is preserved or not. The rest of the section discusses these issues in deeper detail and presents some sophisticated reconfiguration mechanisms, starting from basic graph rewrite rules, to more expressive recursive rules.

4.1 Ad-hoc reconfigurations

An eminent way to formalize graph transformations is the use of graph rewrite rules. This can be done in several flavours [21], but basically the rules come with a left-hand side graph G_L and with a right-hand side graph G_R . Operationally, the rewrite can be applied to any graph G larger than G_L by finding a suitable match (i.e. an occurrence of G_L in G) and the result is the graph obtained from G by removing that instance of G_L and releasing a fresh instance of G_R . Moreover, there can be items shared by G_L and G_R that are required to trigger the rewrite, but are just preserved by the transformation (some sort of interface, needed to properly attach the fresh copy of G_R to the existing items in G).

While this approach is sufficient to model structural aspects, considering QoS requires to enrich graph transformations with constraints. First, graph transformations are equipped with sets of *ask* and *check* constraints. A graph transformation requires the store of the current graph to entail the *ask* constraints and the result of the derivation to be consistent with the *check* constraints. The derived graph has a store that results from first deleting the constraints corresponding to the deleted edges and then adding the *tell* constraints of the rule, i.e., the constraints associated to the new edges.

Consider the problem of dealing with an underloaded ring via reconfiguration rules that act directly at the concrete architecture level. We need rules that take

a single agent from the first ring, detach it and move it to the second ring, also retracting and creating suitable constraints accordingly. Several rules are required because we need to distinguish whether the constraints relate the agent to the agent on its left or its right, and we also need a rule to distinguish the migration of the last agent, which requires to remove the gateway. Of course one might want to specify reconfigurations that move an arbitrary number of agents, hence in general the reconfiguration would consist of a sequence of very specific transformations.

Such a solution, though possible, has some severe drawbacks. First, reconfiguration is not defined as a single action. Second, it is defined at an abstraction level that is far away from the what one has in mind. It would be desirable to define reconfigurations as a single action acting at the most appropriate abstraction level, i.e., rewrite rules should involve both class element and architectural elements, according to the hierarchy fixed by the design.

4.2 Hierarchical reconfigurations

Consider the reconfiguration needed to deal with overloaded rings. Figure 8 depicts a rule that tackles the problem by detaching a chain C_1 from an existing ring to form a new ring on its own with a fresh gateway g . The rule has no *ask* constraint.

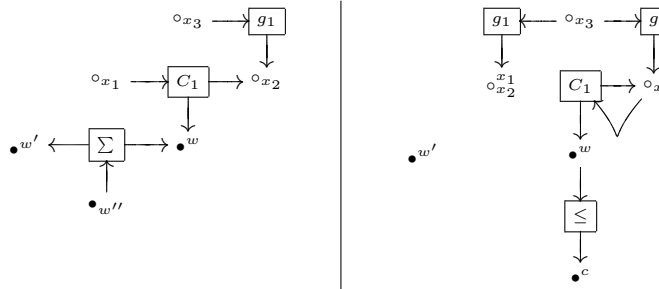


Fig. 8. Reconfiguration rule splitting a ring.

Exploiting the design class *Chain* we can write a unique rule for handling the splitting of arbitrarily large rings.

We have already stated that using architectural styles have several benefits. One of such benefits is that they might offer guarantees about the execution of the system. If one wants to preserve such benefits, then styles should be preserved during run-time. A naive way to guarantee style preservation is to check the style of the architecture after the application of a reconfiguration. This solution, however, is not desirable since there is no guarantee that a necessary style-preserving reconfiguration is possible.

Hence, our methodology has style preservation as a requirement. We need thus a mechanism that ensures style preserving *a priori*, i.e., a mechanism that ensures that any application of a reconfiguration will preserve the architectural style.

Now, while we can be convinced that the rule in Figure 8 preserves the style, this is not obvious when observing the rule itself because we cannot assign style types to the left and right-hand side, because they are incomplete diagrams. We shall see that enforcing style preservation will allow us to use a neater algebraic notation for reconfigurations.

4.3 Style-preserving reconfigurations

There is a very simple sufficient condition for enforcing style preservation, namely that both the left-hand side L and the right-hand side R of the reconfiguration can be assigned the same proper abstract class. Roughly, this way we are guaranteed that whenever L occurs, R would be also allowed by the style. Moreover, since L and R can contain themselves class elements, we are guaranteed that such elements can be consistently refined by any actual design compliant to those classes, keeping the generality of hierarchical reconfigurations.

Consider Figure 9 which proposes a slightly more explicit context than the rule of Figure 8. Apart from minor differences, the key issue is that we have completed the picture of the ring to be transformed, considering also those elements that are not moving to the new ring. In this manner, it is easy to see that both sides are of type N . Indeed, both the left- and right-hand sides of the rule can be derived from a network attached to node x_3 . In term rewriting notation the depicted rule is

$$\text{net}(\text{ring}(\text{chain}(C_1, C_2))) \Rightarrow \text{nets}(\text{net}(\text{ring}(C_2)), \text{net}(\text{ring}(C_2)))$$

Briefly, the rule requires a ring to be decomposable as two chains C_1 and C_2 , respectively. Each chain will form its own ring, by possibly recycling some of the components. Because we assume that reconfigurations are consistent both rings are guaranteed to have enough capacity.

Another instance of style-preserving reconfiguration is the rule depicted in Figures 10 and defined by

$$\text{nets}(\text{net}(\text{ring}(C_1)), \text{net}(\text{ring}(C_2))) \Rightarrow \text{net}(\text{ring}(\text{chain}(C_1, C_2)))$$

It describes a reconfiguration for dealing with two underloaded rings in the following manner: The left ring and right rings are seen as cyclic chains C_2 and C_1 , respectively, attached to the center of the star via their respective gateways. The *ask* constraints (not depicted) are $c' \leq c$, $w' \leq 0.25 \cdot c'$ and $w \leq 0.25 \cdot c$ which respectively guarantee that the new ring will have the best capacity, and both rings are effectively underloaded. Now, merging the rings means removing one of the gateways, connecting the chains to form a cycle and adapting the

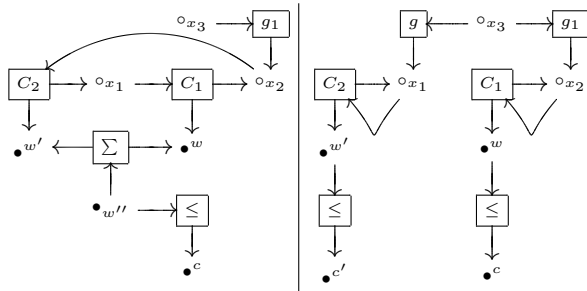


Fig. 9. Reconfiguration rule splitting a ring.

constraints accordingly. It is worth noting that this rule completely abstracts from the internal structure of the chains.

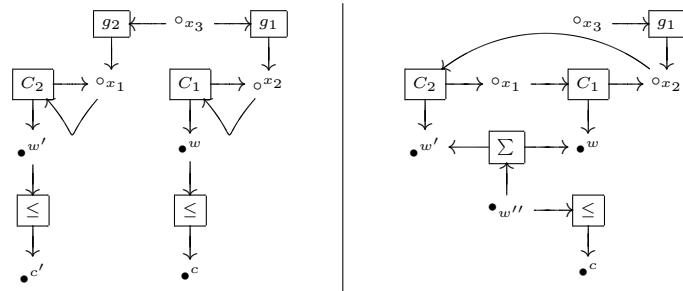


Fig. 10. Reconfiguration rule joining two rings.

4.4 Reconfigurations as term rewriting

We have seen that design rules can be given an algebraic formulation in terms of many-sorted operations over a suitable algebra of typed graphs (with interfaces), with terms describing a particular style-proof. Note that in this way it is possible that: (i) the same well-defined architecture can be described by different terms; (ii) the same well-defined architecture can be assigned different classes.

Since style-preserving reconfigurations essentially operate at the level of style-proofs (the abstract elements in L and R can be seen as typed variables), the algebraic view can be pushed further by term rewriting over (style-)proof terms: a graph transformation rule is seen as a rewrite rule $L \Rightarrow R$, where L and R are \mathcal{M} -terms of the same type. Typically, both L and R are linear and all the variables in R appear in L . This is the case when reconfigurations do not add

abstract components to the system. These variables can be instantiated in any way consistent with the types, and both R and L can be contextualized.

For example, the reconfiguration rule for merging two underloaded rings (cf. Figure 10) can be written just as the following term rewrite whose left-hand side and right-hand side have type N :

$$\mathbf{nets}(\mathbf{net}(\mathbf{ring}(C_1)), \mathbf{net}(\mathbf{ring}(C_2))) \Rightarrow \mathbf{nets}(\mathbf{ring}(\mathbf{chains}(C_1, C_2)))$$

Then, it is possible to apply the rule in any larger well-defined architecture $t(L\eta)$, where η assigns proof terms of type C to variables C_1 and C_2 (for example the elementary chain term **agent**) and where t is any term with one hole of type N (for example, the system proof **sys**(N_1)). After the reconfiguration, the architecture $t(R\eta)$ is obtained.

It is important to remark that in our setting reconfigurations happen at the level of style proofs. Several scenarios are thus possible: (i) the architectural information is available at run-time and it is exploited in the reconfiguration; (ii) the architectural information is available at run-time but we want (or need) to construct a different proof in order to apply the reconfiguration; (iii) no architectural information is available and we need to construct a proper proof in order to apply the reconfiguration.

Case (i) is rather straightforward and computationally less expensive, with the advantage that even in a largely distributed system, all candidates to the reconfigurations can be statically marked and locally monitored. It is however less flexible, because when different reconfigurations are possible that are dependent on different proofs for the same architecture, then no optimization is possible. Instead, case (ii) and (iii) require some additional global monitoring but can allow to improve the performance of the reconfigured system.

4.5 Architectural axioms

The actual design of an architecture is sometimes implicitly embedded in the architecture itself. Consider for instance the tree-like structure of the constraints relating the workloads of the agents forming a ring. Clearly, they reflect the way the ring was constructed by repeatedly concatenating chains of agents. This is not always convenient, since it can limit the applicability of reconfiguration rules.

For instance, while the rule for dealing with an underloaded ring (cf. Figure 10) can be applied to two rings no matter how they were constructed, this is not the case for the splitting rule. Suppose that during the execution the ring obtained by a previous join becomes overloaded and the rule of Figure 9 is chosen for reconfiguration. There is only one way to split the ring, precisely resulting in the same two original rings, because Σ -labeled constraint edges form a statically wired tree and the reconfiguration rule performs the splitting at the top of such tree. This means that whether the architectural information is available at run-time or not, we have at most one way to split any given ring.

This is not a good choice because it might result in badly-balanced rings. Avoiding this in general is not easy. It is up to the architect to define design

rules that reduce the amount of design-dependency. When this is not possible, reconfigurations that restructure the architecture can be defined. For instance, in our example, a better reconfiguration strategy would be to maintain the tree-like structure of chains well balanced before the splitting, so that the reconfigured system is less likely to need further splitting soon.

Consider the reconfiguration in Figure 11 that would allow to re-balance the tree of Σ constraints before the splitting. We may notice that such reconfiguration enjoys three nice properties: (i) it does not influence the topology of the system, because it reconfigures just constraint edges; (ii) it does not modify the global store, because equivalent constraints are expressed on the interface before and after the reconfiguration. Moreover, note that to use the rule for re-balancing (iii) it is better to make it applicable in both directions.

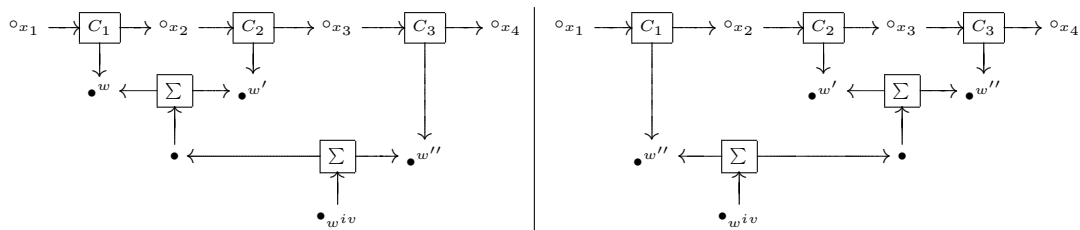


Fig. 11. Architectural axiom for balancing a ring

To some extent, such reconfigurations define *architectural axioms* between two equivalent representations of the same architecture, because the concrete distinctions between the two views L and R are inessential. This scenario is well known in the area of term rewriting, where rewriting up-to equational axioms has received lots of attention, and our rule could be expressed by the axiom:

$$\text{chains}(\text{chains}(C_1, C_2), C_3) \equiv \text{chains}(C_1, \text{chains}(C_2, C_3))$$

The practical consequence, is that the algebra of graphs that we are considering should not distinguish between architectures that have the same structure and are subject to equivalent QoS constraints even when the latter are formulated in slightly different manners.

4.6 More sophisticated reconfigurations

The reconfigurations seen so far can be applied in any larger context with their parameters instantiated according to the specified types. However, it is often the case that a composed architecture can be reconfigured only if all its sub-components are suitably reconfigured first. Stretching the analogy between reconfigurations and rewrite systems the expressiveness of our reconfiguration lan-

guage can be further increased by considering conditional rewrites and labelled rewrites, which are well-established formalisms in the process calculi area.

The first extension allows to limit the applicability of a reconfiguration to certain specific contexts, so that a complex architecture is reconfigured. The second extension allows to tag different families of rewrites depending on their role. Due to space limitation, here we just give a short account of the way in which the expressiveness and usability of our reconfiguration language would be improved by the use of conditional rewrites.

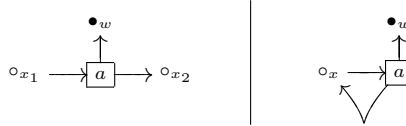
Conditional rewrites take the form

$$\frac{A_1 \rightarrow A'_1 \quad \dots \quad A_n \rightarrow A'_n}{L \rightarrow R}$$

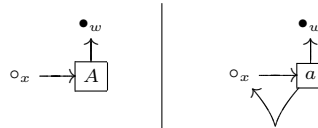
, where L takes A_1, \dots, A_n as parameters, while R takes A'_1, \dots, A'_n . The meaning is that, given an assignment η of concrete architectures to the parameters $A_1, \dots, A_n, A'_1, \dots, A'_n$, the architecture $L\eta$ can be reconfigured according to $R\eta$ only if each $A_i\eta$ can be reconfigured to $A'_i\eta$.

Consider a reconfiguration rule that aims to change the ring topology of basic networks into a star topology, where all agents are connected to the same node. Since we need to reconfigure in one step all the agents of the ring, we cannot provide just one ordinary style-preserving rule. Instead we can give an easy recursive definition via conditional rewrites as below.

The base case consists of coalescing the two ports of an agent (which is a term of type C). This is obtained by the (unconditional) rule **agent** \rightarrow **close(agent)** which is graphically depicted as follows:



The new design production **close** has type $C \rightarrow A$, where A cannot be C since both its interface nodes would be mapped to the same node. Indeed, we introduce a new type A which represents agents attached to one node only. The operation is graphically described by:

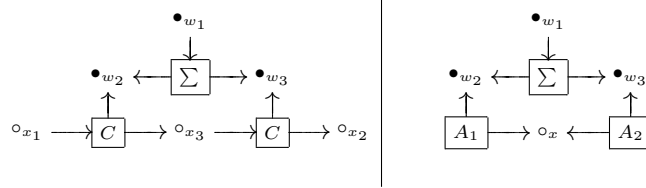


Obviously, because the type C is not preserved by this reconfiguration, the right and left-hand sides of the rewriting rule cannot be applied in the same contexts. This is not a problem but requires to introduce new productions to obtain consistent type reconfigurations. This will be clear with the example.

We now come to the first inductive case which is the transformation of two connected chains into part of a star. The rule is conditional:

$$\frac{C_1, C_2 : C \quad A_1, A_2 : A \quad C_1 \rightarrow A_1 \quad C_2 \rightarrow A_2}{\text{chain}(C_1, C_2) \rightarrow \text{join}(A_1, A_2)}$$

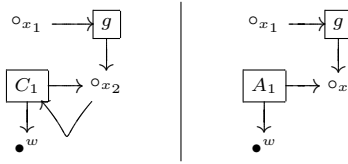
, where production **join** has type $A \times A \rightarrow A$ and its definition is similar to that of operation **chain** (thus we neglect it). We now graphically represent the rewriting rule just defined:



Finally, a ring is star-reconfigured in such a way that the central node of the star is attached to the gateway.

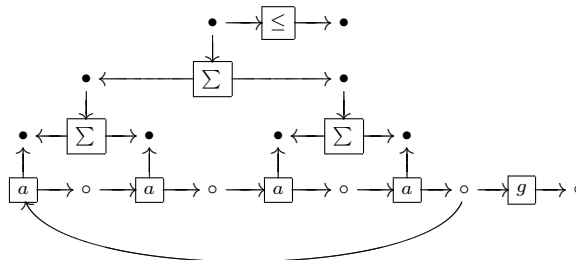
$$\frac{C_1 \quad A_1 \quad C_1 \rightarrow A_1}{\text{ring}(C_1) \rightarrow \text{star}(A_1)}$$

Graphically we have:

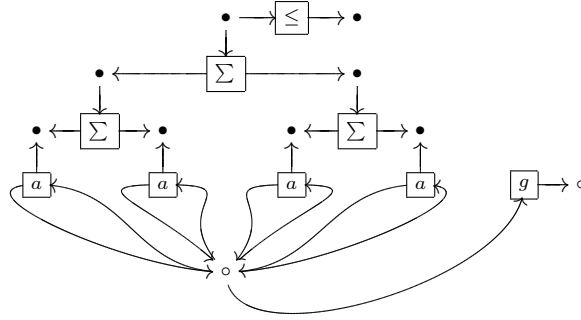


It is worth observing that the desing production **star** has type $A \rightarrow N$, i.e. its codomain is exactly as **ring** (and again, its definition is omitted since it is similar to that of **ring**). This means that we cannot reconfigure a part of a ring, but a whole ring into a star, and networks and systems are guaranteed to preserve their style.

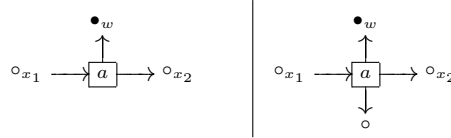
For instance, consider the following system (still of type S).



By applying the reconfiguration just defined we can transform it into:



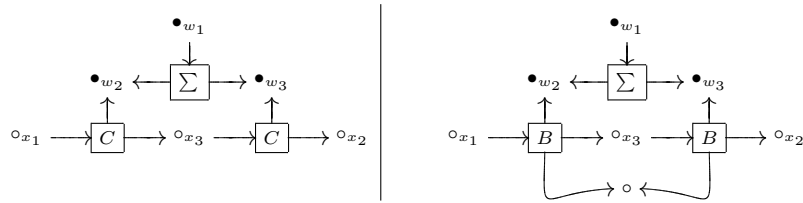
Another version of the example is one in which the agent itself reconfigures by creating a new node that is intended to be the center of the star. We let such an agent be given by desing term **agent***. Thus the base rewrite rule is **agent** \rightarrow **agent*** which is graphically depicted as follows:



We let the type of such an agent to be B (which has three ports in its interface). The next rules are very much like those of the previous example. Thus we shall abuse of notation and overload the name of the operations.

$$\frac{C_1, C_2 : C \quad B_1, B_2 : A \quad C_1 \rightarrow B_1 \quad C_2 \rightarrow B_2}{\text{chain}(C_1, C_2) \rightarrow \text{join}(B_1, B_2)}$$

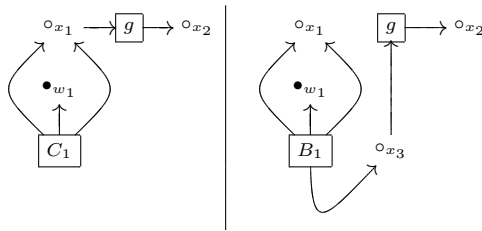
The above rewriting rule is graphically represented by:



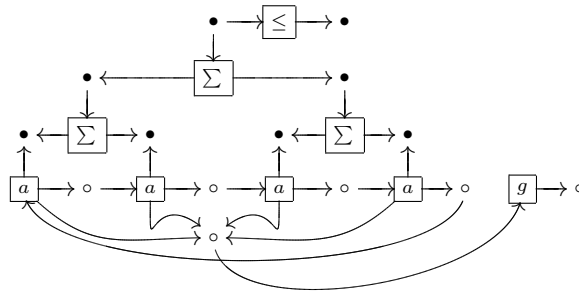
Finally, a ring is star-reconfigured by binding the gateway to the center of the star.

$$\frac{C_1 : C \quad B_1 : B \quad C_1 \rightarrow B_1}{\text{ring}(C_1) \rightarrow \text{star}(B_1)}$$

Graphically we have:



Now, applying the reconfiguration as we did with the previous example we obtain



5 Conclusion

We have presented *Architectural Design Rewriting*, an approach to deal with hierarchical style-based reconfigurations of software architectures with structural and QoS aspects based on a simple algebra of typed graphs with interfaces and constraints, which allows for a unifying treatment of style-based design, style checking, normal execution and reconfiguration.

We believe that our approach has multiple applications. First, it could serve as a formal basis for extending existing methodologies or ADLs for dealing with architectural styles. Then, our approach can also be applied to obtain graphical encodings of nominal calculus with constraints like cc-pi [7].

In future work we plan to analyze and eventually enrich our approach to support the design and management of service-oriented architectures, where design concepts include service assembly and composition [10] and reconfigurations deal with modes [16] and service invocation, discovery and binding [6].

We also plan to perform style analysis as in [19], possibly by applying techniques for the analysis of graph transformation systems (see [2] and the references therein). A closely related issue is to use the logics on graphs presented in those approaches and others that focus on graph logics with QoS aspects (e.g. [9]) in order to impose additional constraints.

6 Acknowledgments

This work has been partly supported by the EU within the FETPI Global Computing, project IST-2005-016004 SENSORIA (*Software Engineering for Service-Oriented Overlay Computers*).

References

1. S. Ansaloni, A. Sztajnberg, R. Cerqueira, and O. Loques. Deploying QoS contracts in the architectural level. In P. Dissaux, M. Amine, and P. Michel, editors, *Architecture Description Languages*, pages 19–34. Springer Verlag, 2004. 18th IFIP World Computer Congress.
2. P. Baldan, A. Corradini, B. König, and A. Lluch Lafuente. A temporal graph logic for verification of graph transformation systems. In *WATD'06*, number 4409 in Lecture Notes in Computer Science. Springer Verlag, 2007.
3. L. Baresi, R. Heckel, S. Thöne, and D. Varró. Style-based modeling and refinement of service-oriented architectures. *Software and Systems Modeling*, 5(2):187–207, June 2006.
4. S. Bistarelli, U. Montanari, and F. Rossi. Semiring-based constraint satisfaction and optimization. *Journal of the ACM*, 44(2):201–236, 1997.
5. S. Bistarelli, U. Montanari, and F. Rossi. Soft concurrent constraint programming. *ACM Trans. Comput. Logic*, 7(3):563–589, 2006.
6. L. Bocchi, J. Fiadeiro, and A. Lopes. Primitives for configuration management. Internal Document, September 2005.
7. M. Buscemi and U. Montanari. CC-Pi: A constraint-based language for specifying service level agreements. In *Proceedings of the 16th European Symposium on Programming (ESOP'07)*. Springer Verlag, 2007. To appear.
8. P. Clements, D. Garlan, L. Bass, J. Stafford, R. Nord, J. Ivers, and R. Little. *Documenting Software Architectures: Views and Beyond*. Pearson Education, 2002.
9. G. Ferrari and A. Lluch Lafuente. A logic for graphs with qos. *Electronic Notes on Theoretical Computer Science*, 142:143–160, 2006.
10. J. L. Fiadeiro, A. Lopes, and L. Bocchi. A formal approach to service component architecture. In M. Bravetti, M. Núñez, and G. Zavattaro, editors, *WS-FM*, volume 4184 of *Lecture Notes in Computer Science*, pages 193–213. Springer, 2006.
11. P. Fradet and D. L. Metayer. Shape types. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 27–39, New York, NY, USA, 1997. ACM Press.
12. D. Garlan, R. T. Monroe, and D. Wile. ACME: an architecture description interchange language. In J. H. Johnson, editor, *CASCON*, page 7. IBM, 1997.
13. D. Garlan and B. Schmerl. Architecture-driven modelling and analysis. In T. Cant, editor, *Proceedings of the 11th Australian Workshop on Safety Related Programmable Systems (SCS'06)*, volume 69 of *Conferences in Research and Practice in Information Technology*, Melbourne, Australia, 2006.
14. I. Georgiadis, J. Magee, and J. Kramer. Self-organising software architectures for distributed systems. In D. Garlan, J. Kramer, and A. L. Wolf, editors, *WOSS*, pages 33–38. ACM, 2002.
15. D. Hirsch, P. Inverardi, and U. Montanari. Modeling software architectures and styles with graph grammars and constraint solving. In P. Donohoe, editor, *WICSA*, volume 140 of *IFIP Conference Proceedings*, pages 127–144. Kluwer, 1999.
16. D. Hirsch, J. Kramer, J. Magee, and S. Uchitel. Modes for software architectures. In V. Gruhn and F. Oquendo, editors, *EWSA*, volume 4344 of *Lecture Notes in Computer Science*, pages 113–126. Springer, 2006.
17. D. Hirsch and U. Montanari. Shaped hierarchical architectural design. *Electronic Notes on Theoretical Computer Science*, 109:97–109, 2004.
18. D. Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering Methodologies*, 11(2):256–290, 2002.

19. J. S. Kim and D. Garlan. Analyzing architectural styles with alloy. In *ROSATEA '06: Proceedings of the ISSTA 2006 workshop on Role of software architecture for testing and analysis*, pages 70–80, New York, NY, USA, 2006. ACM Press.
20. N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000.
21. G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1997.
22. V. A. Saraswat. *Concurrent constraint programming*. MIT Press, Cambridge, MA, USA, 1993.
23. M. Shaw and D. Garlan. *Software Architectures: Perspectives on an emerging discipline*. Prentice Hall, 1996.