# Constraint Design Rewriting

Roberto Bruni[a], Alberto Lluch Lafuente[b], Ugo Montanari[a]

[a] *Dipartimento di Informatica, Università di Pisa, Italy*
{bruni,ugo}@di.unipi.it

[b] *IMT Institute for Advanced Studies Lucca, Italy*
alberto.lluch@imtlucca.it

**Abstract**

Constraint networks are hyper-graphs whose nodes and hyper-edges represent variables and relations between them, respectively. The problem to assign values to variables by satisfying all constraints is NP-complete. We propose an algebraic approach to the design and transformation of constraint networks, inspired by *Architectural Design Rewriting* (ADR). The main idea is to exploit ADR to equip constraint networks with some hierarchical structure and represent them as terms of a suitable algebra, when possible. Constraint network transformations such as *constraint propagations* are then specified with efficient rewrite rules exploiting the network's structure provided by terms. The approach can be understood as (i) an extension of ADR with constraints, and (ii) an application of ADR to the design of reconfigurable constraint networks.

*Keywords:* Constraints, Rewriting, Hierarchical Graphs, Architectures

It is a pleasure for all of us to contribute this piece of work in honor of Paul Klint. I always considered his role at CWI as essential in providing an excellent software engineering counterpart to the other theoretical computer science components of the Center. In my long research life I had the occasion of working also on software engineering issues, and I very much appreciated Paul's contributions to the field. Also, I consider EAPLS, which for some time I represented at ETAPS Steering Committee, as an important achievement by Paul Klint. I found his message announcing EAPLS' foundation:

```
On Dec 4, 1996, at 10:25 AM, Paul Klint wrote:

Dear EAPLS enthusiasts:

Following the EAPLS meeting in Aachen last September, EAPLS has now been officially founded!
The formal documents were signed last October 24, in Amsterdam.
```

Ugo Montanari (Pisa, August 2013)

## 1. Introduction

Constraint networks [1, 2] are a very flexible and general formalism used to model and solve a wide variety of applications such as optimization problems, knowledge representation, and synchronization mechanisms, to mention a few [3]. Technically, constraint networks are hyper-graphs whose nodes and hyper-edges are respectively interpreted as variables and relations constraining the assignment of values to the variables of their adjacent nodes. Typically, the hyper-graph represents a system composed by several entities represented by hyper-edges that are inter-connected with each other by attaching their tentacles to shared nodes. Such entities may be, for instance, software artifacts such as software components within an architecture or classes within a class diagram. The use of constraints has

Research supported by the European projects IP 257414 ASCENS and STReP 600708 QUANTICOL, and the Italian PRIN 2010LHT4KM CINA.

several practical uses. For instance, it allows software artifacts to delay the actual choice of values associated to their connections (e.g. the actual choice of the bandwidth to be allocated on a channel) and thus facilitate the development of open-ended systems made of loosely coupled artifacts (e.g., autonomous or service-oriented systems) which may connect by reaching an agreement on the admissible values on shared resources at run-time. As a matter of fact, the problem of finding all possible agreements is the most typical and studied problem for networks of constraints, called *Constraint Satisfaction Problem* (CSP), which consists more precisely on determining all the assignments of values to variables which satisfy all constraints. These problems are NP-complete, thus they cannot be solved efficiently in general. Special cases allowing for feasible solutions have been sought actively in the Artificial Intelligence field in the past forty years. Especially useful is the *perfect relaxation* method [2], based on dynamic programming. The idea is to find a derivation, namely a syntax tree, for the given network using a hyper-edge replacement grammar whose productions are *small* (in terms both of the number of tentacles of the hyper-edge in their left members, and in the size of the graphs in their right members). Then the solution to the original problem is decomposed into a sequence (or, rather, a tree) of smaller problems, one for every step in the derivation: considering the grammar rule used in that step, the CSP problem for the graph in the right hand side is solved and the resulting relation is assigned to the hyper-edge in the left side, to be recursively employed in a bottom up fashion in the next step. This algorithm is *linear* within the class of constraint networks whose underlying graph is generated by a (finite) hyper-edge replacement grammar.

Architectural Design Rewriting (ADR) [4] is a formal approach to the design of reconfigurable software systems. ADR offers a formal setting where design development, run-time execution and reconfiguration aspects are defined on the same footing. One of the main features of ADR is the ability to characterize a class of graphs satisfying certain spatial constraints by means of a graph algebra. The flexibility of ADR is evidenced by its many applications to several aspects of software engineering, including model driven transformations [5], architectural styles and reconfigurations [4, 6, 7], modeling of service oriented systems [8, 9], and graphical representation of process calculi [10]. As mentioned for the case of networks of constraints, it is very often the case that hyper-edges represent software artifacts. The spatial constraints that ADR allows one to specify and exploit consist then in the allowed topological ways of connecting those artifacts, typical cases being metamodels and architectural styles.

In this paper we present some preliminary ideas on how to combine some techniques from ADR and from constraint networks. Our proposal can be understood both as (i) an enrichment of ADR with non-spatial constraints, and (ii) an application of the ADR methodology to the design and transformation of structured constraint networks. The main idea is to model classes of constraint networks as algebras, whose operators can be used to denote constraint networks with terms. Network transformations, like constraint propagation, are then specified by rewrite rules that exploit the structure provided by terms.

One of the key issues is that ADR graphs can be hierarchical and, indeed, the ADR graph algebra [10] has primitive operations to encapsulate a graph within a box with tentacles (a hyper-edge). The resulting structure is compositional in two dimensions: (i) hyper-edges and nodes can be connected to obtain ordinary graphs using operators reminiscent of parallel composition and restriction of process algebras; (ii) the encapsulation operation can conveniently model an abstraction/refinement step of the design. In particular, if a graph grammar based on hyper-edge replacement [11] is employed to define an architectural style [12], an ADR graph is able to model not only a resulting (style-compliant) architecture, but also its syntax tree, recording all refinement steps of the design process.

The ability to represent both a graph and its syntax tree is particularly relevant for constraint networks [1, 2]. It is now clear why ADR graphs are convenient for modeling networks of constraints: not only does hierarchical structure record the steps of the design process, but also the same structure is essential at run-time for efficiently checking the satisfiability of the resulting global constraint. ADR also facilitates the seamless handling of network reconfiguration defined by structural induction, a feature not considered in [13] and that is needed when the architectural style (i.e., the selected hyper-edge replacement grammar) is changed at run-time. Also, when the more general case of Constraint Logic Programming (CLP) is considered [13], and a satisfiability check is required at every step, the condition about the underlying graph being derivable by a hyper-edge replacement grammar turns out to be automatically satisfied. The promotion of ADR for supporting the design and evaluation of constraints is the main contribution of this paper.

Our approach aims to combine Computer Science principles that are frequently used for the purpose of *Understanding Software*: namely *abstraction* (as provided by the use of interfaces and hierarchies), *compositionality* (as provided by algebras and grammars), *structure* (as provided by terms and graphs), *visual representation* (as provided by graphs), *partial information* (as provided by constraints), and *declarative specification* (as provided by rewrite rules and constraints).

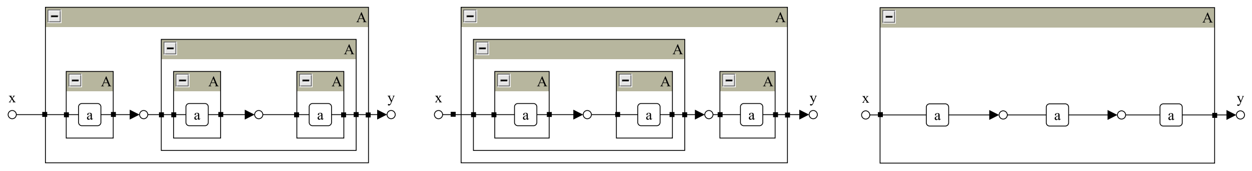

Figure 1: Two hierarchical designs (left, middle) and their flattened version (right).

## 2. Constraint Design Rewriting

In this section we will first present an algebraic notation (§ 2.1) for networks of constraints [2]. Next (§ 2.2) we will explain how to exploit the algebraic presentation for providing an efficient mechanism for constraint solving. Due to space limitation, we will use a very simple running example inspired on the well-known *pipes-and-filters* architectural style. It requires software components within an architecture to be composed as a connected sequence, that is, loops, branches or disconnected parts are not allowed. Each component acts as an information processing unit that filters or transforms the information pieces it receives on its input port and delivers them on its output port. The level of security of the information is to be increased, i.e., the level of information of the input is smaller than the level of information of the output. We shall see how ADR and constraints facilitate the design of such an architecture.

### *2.1. Constraint Design*

Networks of constraints [2] are essentially (hyper-)graphs together with an interpretation of nodes as variables and (hyper-)edges as relations between admissible values on the attached nodes. We give an algebraic presentation of networks of constraints that we call *constraint network designs*, which allow us to denote constraint networks as terms over a suitable syntax.

*ADR Designs.* Networks of constraints have a structural part (i.e., the network) that essentially amounts to (possibly hierarchical) graphs with interfaces. We will use ADR designs [4] to model them and the design algebra of [10] (an extension of the graph algebra of [14]) to build them.

**Definition 2.1 (design).** *A* design *is a term of sort* $\mathbb{D}$ *defined by the grammar*

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

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

The syntax has the following informal meaning: $\mathbf{0}$ is the empty graph, $x$ represents a discrete graph containing node $x$ only, $l\langle\overline{x}\rangle$ is a graph formed by an $l$-labelled hyper-edge attached to nodes $\overline{x}$ (the $i$-th tentacle is attached to the $i$-th node in $\overline{x}$), $\mathbb{G} \parallel \mathbb{H}$ is the graph resulting from the disjoint union of graphs $\mathbb{G}$ and $\mathbb{H}$ up to shared nodes (i.e., nodes with the same names are merged, but edges are never merged). The operators $\mathbf{0}$, $x$, $l\langle\overline{x}\rangle$ and $\mathbb{G} \parallel \mathbb{H}$ suffice to represent any hyper-graph (and any non-structured constraint network).

Sometimes the exact names of certain nodes are not important. For example, it is frequent to deal with large hyper-graphs where only a small set of nodes is meaningful for gluing further edges. We can abstract away from the name $x$ of a node by using the operator $(\nu x)\mathbb{G}$, which represents the graph $\mathbb{G}$ after making node $x$ *restricted*, i.e., not visible from the outside. The non-specialist can think of $x$ as a local variable in $\mathbb{G}$. For example, while the term $a\langle x, y\rangle \parallel a\langle x, y\rangle$ represents a graph with two nodes $x$ and $y$ and two $a$-labelled edges attached to them, the term $\mathbb{G} \parallel \mathbb{G}$, where $\mathbb{G} = (\nu x)a\langle x, y\rangle$, represents a graph with three nodes (one node called $y$ and two local versions of the nodes called $x$) and two edges (each directed from a different local version of $x$ to the same node $y$).

Edge-encapsulated graphs are called designs. A term $L_{\overline{x}}[\mathbb{G}]$ is a design labelled by $L$, with body graph $\mathbb{G}$ whose nodes $\overline{x}$ define the "interface" of the encapsulating $L$-labelled edge.

Hierarchical hyper-graphs are then represented thanks to the operator $\mathbb{D}\langle\overline{x}\rangle$ that allows one to attach a design $\mathbb{D}$ to nodes $\overline{x}$ (the $i$-th node in the interface of $\mathbb{D}$ to the $i$-th node in $\overline{x}$). In the following we say that a term is *hierarchical* if it contains designs, otherwise we call it *flat*.

The non-specialist can think of a design $L_{\overline{x}}[\mathbb{G}]$ like a procedure declaration, where $\overline{x}$ is the list of formal parameters. Then, the term $L_{\overline{x}}[\mathbb{G}]\langle\overline{y}\rangle$ represents the application of the procedure to the list of actual parameters $\overline{y}$; of course, in this case the lengths of $\overline{x}$ and $\overline{y}$ must be equal.

The algebraic reading of the grammar in Definition 2.1 is as usual, where each syntactic category and vocabulary is considered as a sort and productions are read as operations. This allows us, for instance, to consider open terms (i.e. terms with sorted variables): they are useful for defining sub-classes of designs (e.g. architectural styles or encodings) by means of derived operators as we shall see in Example 2.2.

Restriction $(\nu x)\mathbb{G}$ acts as a binder for $x$ with scope $\mathbb{G}$ and similarly $L_{\overline{x}}[\mathbb{G}]$ binds names $\overline{x}$ with scope $\mathbb{G}$. Restrictions and interfaces lead to the usual notion of *free* nodes, denoted by the function $fn(\cdot)$. The set of all (distinct) nodes in a graph or design are denoted with $n(\cdot)$. In the following, we let $\lfloor\overline{x}\rfloor$ denote the set of elements $\overline{x}$.

Edge and design labels are assigned an arity (i.e., the number of their tentacles) that we assume to be respected, i.e., designs and graphs are *well-typed*. A well-typed design or graph is *well-formed* if (i) for each occurrence of design $L_{\overline{x}}[\mathbb{G}]$ we have $\lfloor\overline{x}\rfloor \subseteq fn(\mathbb{G})$; and (ii) for each occurrence of graph $L_{\overline{x}}[\mathbb{G}]\langle\overline{y}\rangle$, the substitution ${}^{\overline{x}}/_{\overline{y}}$ is a function. We restrict our attention to well-formed designs.

The algebra includes suitable structural axioms, like associativity and commutativity for $\|$ with identity $\mathbf{0}$ and name extrusion and axioms to $\alpha$-rename bound nodes, which define an equivalence relation $\equiv_{\mathrm{D}}$ on terms. The main result of [10] shows that the equivalence classes of terms are in bijective correspondence with a set-theoretic definition of hierarchical hyper-graphs.

**Example 2.1.** *Recall our running example of an architecture of information filters, whose components have one input and one output port. Each component is modeled as a hyper-edge with two tentacles, modeling the input and output ports, respectively. We use $a \in \mathcal{E}$ to label atomic filters (i.e. edges), $A \in \mathcal{D}$ to label hierarchical filters (i.e. designs), and $x, y, z, u, v, w \in \mathcal{N}$ to denote port attachments (i.e. nodes). Figure 1 depicts three designs representing three different filter architectures. Those designs are denoted by terms $\mathbb{D}_1$ (left), $\mathbb{D}_2$ (middle) and $\mathbb{D}_3$ (right) defined below:*

$$\begin{array}{lcl}
\mathbb{D} & = & A_{x,y}[a\langle x,y\rangle] \\
\mathbb{D}_1 & = & A_{x,y}[\,(\nu z)\,(\,\mathbb{D}\langle x,z\rangle \;\|\; A_{u,v}[\,(\nu w)(\mathbb{D}\langle u,w\rangle \,\|\, \mathbb{D}\langle w,v\rangle)\,]\langle z,y\rangle\,)\,] \\
\mathbb{D}_2 & = & A_{x,y}[\,(\nu w)\,(\,A_{u,v}[(\nu z)(\mathbb{D}\langle u,z\rangle \,\|\, \mathbb{D}\langle z,v\rangle)\,]\langle x,w\rangle \;\|\; \mathbb{D}\langle w,y\rangle\,)\,] \\
\mathbb{D}_3 & = & A_{x,y}[(\nu z,w)(a\langle x,z\rangle \,\|\, a\langle z,w\rangle \,\|\, a\langle w,y\rangle)]
\end{array}$$

*Nodes are represented by circles, edges by small rounded boxes, and designs by large shaded boxes with a top bar. The first tentacle of an edge (i.e., the input port) is represented by a plain arrow with no head, while the second one (i.e., the output port) is denoted by a normal arrow. If a node is exposed in the interface we put it on the outermost layer and overlap the edges of the various layers denoting this with black boxes on design borders.*

Hierarchical designs (graphs) can be flattened into flat designs (graphs) by means of *flattening* rules. They take the form $L_{\overline{x}}[\mathbb{G}]\langle\overline{y}\rangle \Rightarrow \mathbb{G}\{{}^{\overline{y}}/_{\overline{x}}\}$, which corresponds to a sort of hyper-edge replacement [11]. Such rules may also be included as axioms of $\equiv_{\mathrm{D}}$ to consider designs up to flattening and nesting. Another useful operation is that of *unfolding* which is very much like flattening but keeping the enclosing edge being flattened as an additional ordinary edge. Unfolding rules are of the form $L_{\overline{x}}[\mathbb{G}]\langle\overline{y}\rangle \Rightarrow \mathbb{G}\{{}^{\overline{y}}/_{\overline{x}}\} \,\|\, L\langle\overline{y}\rangle$. Finally, we shall consider the *abstraction* operation that transforms a design in an ordinary edge. Abstraction rules are of the form $L_{\overline{x}}[\mathbb{G}]\langle\overline{y}\rangle \Rightarrow L\langle\overline{y}\rangle$. The bottom-up application of abstraction rules until a fixpoint is reached defines an abstraction function $\widehat{\cdot}$.

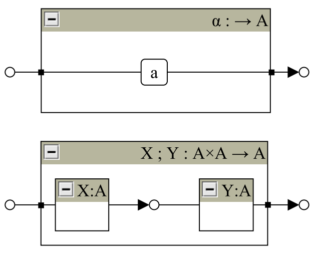

Figure 2: An algebra of sequences.

*ADR Design Styles.* Classes of graphs with particular shapes can be defined by graph language formalisms such as graph grammars. The approach to graph classes in ADR is based on an algebraic interpretation of context-free graph grammars (hyper-edge replacement graph grammars): non-terminal symbols (design labels) are interpreted as sorts and productions are interpreted as operations whose domain and co-domain are given by the non-terminal symbols in the left- and right-hand side of the production, respectively.

A *design style* is a tuple $\mathcal{S} = \langle \Sigma, \mathsf{E}, \mathsf{A}\rangle$ where $\Sigma$ is the signature of the style (sorts, labels, operation symbols), $\mathsf{E}$ is the set of equations that define the operations (as derived operators of the design algebra) and $\mathsf{A}$ is a set of axioms preserving $\equiv_{\mathsf{D}}$ ($\mathbb{D}_1 \equiv_{\mathsf{A}} \mathbb{D}_2$ implies $\mathbb{D}_1 \equiv_{\mathsf{D}} \mathbb{D}_2$). Choosing $\mathsf{A}$ such that $\equiv_{\mathsf{A}}$ exactly coincides with $\equiv_{\mathsf{D}}$ is convenient since one obtains an exact correspondence between graph and term rewriting (graph matching based on subgraph isomorphism corresponds to term matching modulo $\mathsf{A}$). We say that a design $\mathbb{D}$ has type $A$, written $\mathbb{D} : A$ whenever $\mathbb{D} \equiv A_{\overline{x}}[\mathbb{G}]$.

The next example illustrates the two roles of the nesting operator: to enclose a graph and as a sort of typed interface to enable disciplined graph compositions.

**Example 2.2.** *The pipes-and-filter architectural style of our running example can be defined on top of the simple style $\mathcal{S}_L = \langle \Sigma_L, \mathsf{E}_L, \mathsf{A}_L \rangle$ that characterises the set of a-labelled, acyclic, and connected sequences. This is needed to exclude, e.g., designs of the form $A_{x,y}[a\langle x, y\rangle \parallel a\langle x, y\rangle]$ (multiple branching) and $A_{x,y}[a\langle x, y\rangle \parallel a\langle y, x\rangle]$ (looping).*

*The operations of the style consists of the constant operator $\alpha :\rightarrow A$ to introduce elements in the sequence (i.e. atomic filters), and a binary sequential composition $\_;\_ : A \times A \rightarrow A$. Both are derived operators defined by the following two equations in $\mathsf{E}_L$: $\alpha = A_{u,v}[a\langle u, v\rangle]$ and $\mathbb{D}_1; \mathbb{D}_2 = A_{u,v}[(\nu w)(\mathbb{D}_1\langle u, w\rangle \parallel \mathbb{D}_2\langle w, v\rangle)]$, where $\mathbb{D}_1, \mathbb{D}_2 : A$.*

*The graphical representation of both operators is visualised in Fig. 2. We put the operator declaration on the top bar of the outermost design and we annotate the variables with their names and types. Note that, implicitly, the type of the outermost box is the type returned by the operation.*

*Clearly, the style as such constructs hierarchical sequences of filters, where, e.g. $\alpha; (\alpha; \alpha)$ (cf. left of Figure 1 and $\mathbb{D}_1$ in Example 2.1) and $(\alpha; \alpha); \alpha$ (cf. right of Figure 1 and $\mathbb{D}_2$ in Example 2.1) define different designs due to the different nestings. If flattening is included as an axiom in $\mathsf{A}_L$ then $\mathsf{A}_L$ should also include the associativity of $\_;\_$ so that $\equiv_{\mathsf{A}_L}$ and $\equiv_{\mathsf{D}}$ coincide. Then, the two former terms are identified (cf. right of Figure 1 and $\mathbb{D}_3$ in Example 2.1).*

*Constraint Network Designs.* Interpreting the nodes of a graph as variables and the arcs as constraints gives rise to a *network of constraints* [2]. In this work, we define a *constraint network design* as a pair $\langle \mathbb{D}, I\rangle$ where $\mathbb{D}$ is a design and $I : (\mathcal{E} \cup \mathcal{D}) \rightarrow P(\mathcal{U}^k)$ is a (rank-respecting) interpretation function mapping edge and design labels to constraints, i.e., $\mathcal{U}$ is the domain of variables (nodes) and $P(\mathcal{U}^k)$ is the set of all $k$-relations on $\mathcal{U}$.

**Example 2.3.** *We can now introduce in our running example the architectural constraints related to the information security level. Let us take the set of natural numbers $\mathbb{N}$ to denote such level to be used as the domain of nodes and take the interpretation function I that maps labels a and A of our running example to $\{(x, y) \in \mathbb{N} \mid x < y\}$. Such interpretation models the fact that filters (atomic and hierarchical) impose a simple constraint on the security level of information produced: it shall be strictly higher than the information received. Abstractly, any constraint network design $\langle \mathbb{D}, I\rangle$ requires a strictly ordered sequence of natural numbers to be assigned to the nodes in $\mathbb{D}$.*

We say that $\langle \mathbb{D}, I\rangle$ is a *hierarchical network design* whenever $\mathbb{D}$ is hierarchical, otherwise we call it *flat network design*. It is easy to see that every (hierarchical) network of constraints can be denoted by a (hierarchical) constraint network design in a manner similar to the way that graphs are denoted by designs in ADR [10]. In addition, we lift the notion of style to *network design style* in the obvious way.

*Constraint Satisfaction Problem.* Solving a network of constraints is called a *constraint satisfaction problem (CSP)*. We recast here the original presentation of [2] in terms of designs.

Let $\langle L_{\overline{x}}[\mathbb{G}], I\rangle$ be a flat constraint network design, set $m = |n(\mathbb{G})|$ and let $\langle n_1, \ldots, n_m\rangle$ be any ordering of the variables (nodes) in $n(\mathbb{G})$. For ease of presentation and without loss of generality we assume that $\mathbb{G}$ is in normal form $(\nu\overline{z})\mathbb{H}$, where $\mathbb{H}$ has no restricted name (and is flat). Then, the solution $sol(\langle L_{\overline{x}}[\mathbb{G}], I\rangle)$ of $\langle L_{\overline{x}}[\mathbb{G}], I\rangle$ is the set $\{\overline{v}_{|\overline{x}} \mid \overline{v} \in \mathcal{U}^m \text{ and } \mathbb{H} \equiv_{\mathsf{D}} b(\overline{y}) \parallel \mathbb{F} \text{ implies } \overline{v}_{|\overline{y}} \in I(b)\}$, where we denote the projection of a vector $\langle n_{i_1}, \ldots, n_{i_k}\rangle$ on $\overline{v} = \langle v_1, \ldots, v_m\rangle \in \mathcal{U}^m$ as $\overline{v}_{|\langle n_{i_1},\ldots,n_{i_k}\rangle} = \langle v_{i_1}, \ldots, v_{i_k}\rangle$. In other words, the CSP for a constraint network is to find the set of all the assignments of the variables $\overline{x}$ connected by the interface arc that can be extended to all the $m$ variables in $n(\mathbb{G})$ by satisfying all the constraints in $\mathbb{G}$ (as imposed by each arc $b\langle\overline{y}\rangle$ according to $I$). We say that a constraint network design $\langle \mathbb{D}, I\rangle$ is *consistent* if $sol(\langle \mathbb{D}, I\rangle) \neq \emptyset$ and that it is *inconsistent* otherwise.

In general, hierarchical networks can be solved by first unfolding them. We shall see that, when the structure of the network is exploited, unfolding is not necessary: the hierarchical structure can be preserved and hierarchical designs can be annotated with the solution of the part of the network they are enclosing, to be exploited to solve the network recursively.

$$\frac{s = sol(\langle \widehat{f}(\mathbb{D}_1, \ldots, \mathbb{D}_n), I\rangle)}{\langle f(\mathbb{D}_1 : S_1^{s_1}, \ldots, \mathbb{D}_n : S_n^{s_n}), I\rangle \to_p \langle f^s(\mathbb{D}_1, \ldots, \mathbb{D}_n), I \cup \{S^s \mapsto s\}\rangle}\ (\mathsf{SOLVE}_f)$$

$$\frac{i \in [1,n] \qquad \langle \mathbb{D}_i, I\rangle \to_p \langle \mathbb{D}'_i, I'\rangle \qquad \forall j \neq i.\ \mathbb{D}'_j = \mathbb{D}_j}{\langle f(\mathbb{D}_1, \ldots, \mathbb{D}_n), I\rangle \to_p \langle f(\mathbb{D}'_1, \ldots, \mathbb{D}'_n), I'\rangle}\ (\mathsf{IND}_f)$$

Figure 3: Perfect relaxation on generic styles.

**Example 2.4.** *In our running example, a design $\langle \mathbb{D} : A, I\rangle$ can be understood as an information filtering architecture where the choice of concrete security levels to be used in port attachments (i.e., nodes) have been delayed. The set of all admissible assignments of security levels to port attachments is precisely the solution to the constraint network design $\langle \mathbb{D} : A, I\rangle$. As a matter of fact, the solution $\langle \mathbb{D} : A, I\rangle$ is $sol(\langle \mathbb{D} : A, I\rangle) = \{(x, y) \in \mathbb{N}^2 \mid x + k \leq y\}$, where $k$ is the number of occurrences of $\alpha$ in $\mathbb{D}$. In other worlds, admissible architectures are such that the security level on the output port is greater than the security level on the input port by a difference equal to the number of internal atomic filters. The solution can be computed using a standard CSP solving procedure, but, as we shall see, one can compute it by exploiting the structure of $\mathbb{D}$.*

*Consistent Network Design Development.* Ideally, constraint network designs should be developed so to guarantee consistency. However, this is a challenging task since very easily the choices of the constraint interpretation of $I$ or the design style may allow the construction of inconsistent networks.

**Example 2.5.** *Let us consider the pipes-and-filter style $\mathcal{S}_L$ of Example 2.3 and suppose that we define an interpretation function $I'$ such that $I'(A) = I(A)$ and $I'(a) = \{(x, y) \in \mathbb{N}^2 \mid x > y\}$. In words, atomic filters impose an inverse constraint with respect to hierarchical filters, decreasing the security level rather than increasing it. Then it is easy to see that any constraint network design $\langle \mathbb{D} : A, I'\rangle$ in constraint network design style $\langle \Sigma_L, \mathsf{E}_L, \mathsf{A}_L, I'\rangle$ is inconsistent.*

*Suppose instead that we define an interpretation $I''$ such that $I''(A) = I(A)$ and $I''(a) = \{(0, 1)\}$. In words we restrict ourself to filters that admit non-secure information and deliver secure information. The new style $\langle \Sigma_L, \mathsf{E}_L, \mathsf{A}_L, I'\rangle$ allows us to build the consistent design $\langle \alpha, I\rangle$, but also the inconsistent design $\langle \alpha; \alpha, I\rangle$ which results by combining together two consistent designs. In other words, a style may not guarantee compositional consistency.*

Ideally, a style $\mathcal{S} = \langle \Sigma, \mathsf{E}, \mathsf{A}, I\rangle$ would consist of operators $f \in \Sigma$ of functional type $f : S_1 \times \cdots \times S_n \to S$ and the interpretation function $I$ would be such that for any given consistent constraint network designs $\langle \mathbb{D}_i : S_i, I\rangle$ we are guaranteed that $\langle f(\mathbb{D}_1, \ldots, \mathbb{D}_n), I\rangle$ is consistent. If this property is shown for all operators $f \in \Sigma$ then, by induction, any possible constraint network design is guaranteed to be consistent. However, designing such a *compositionally consistent* style may not be easy. Indeed, the style of Example 2.3 does not satisfy this property. The next example shows a compositionally consistent style.

**Example 2.6.** *If we would like to have a compositionally consistent style for our running example we could proceed by defining a variant of $\mathcal{S}_L$ as follows. The signature includes a family of parametric edge labels $a^{[i,j]}$ and design labels (sorts) $A^{[i,j]}$ (for $i < j$), and a subsorting relation $A^{[i,j]} \leq A^{[k,l]}$ whenever $k \leq i$ and $j \leq l$. The idea is that an $A^{[i,j]}$-labelled sequence contains natural numbers in the range $[i, j]$. The subsorting relation captures the idea that one can always replace a design by another one weaker constraints. The operations of our style are parametric as well, with families of operators $\alpha_{i,j} :\to A^{[i,j]}$ and $\_;\_ : A^{[i,j]} \times A^{[k,l]} \to A^{[i,l]}$ (defined only when $j \leq k$), respectively defined as: $\alpha^{[i,j]} = A^{[i,j]}_{u,v}[a^{[i,j]}\langle u, v\rangle]$ and $\mathbb{D}_1; \mathbb{D}_2 = A^{[i,l]}_{u,v}[(vw)(\mathbb{D}_1\langle u, w\rangle \parallel \mathbb{D}_2\langle w, v\rangle)]$, where $\mathbb{D}_1 : A^{[i,j]}$ and $\mathbb{D}_2 : A^{[k,l]}$. The interpretation function $I$ is such that $I(A^{[i,j]}) = I(a^{[i,j]}) = \{(x, y) \in \mathbb{N} \mid i \leq x < y \leq j\}$, i.e. each sort imposes the maximum (resp. minimum) security level of the input (resp. output) port.*

## 2.2. Constraint Rewriting

Rewriting techniques can be used to specify constraint network transformations in a declarative way. A typical example is that of *constraint relaxation* which provides a simple and efficient mechanism for constraint solving.

$$\frac{s = sol(\langle \widehat{\alpha}, I\rangle)}{\langle \alpha, I\rangle \rightarrow_p \langle \alpha^{s_1}, I \cup \{A^s \mapsto s\}\rangle}\ (\mathsf{SOLVE}_\alpha)$$

$$\frac{s = sol(\langle \mathbb{D}_1 \mathbin{\widehat{;}} \mathbb{D}_2, I\rangle)}{\langle \mathbb{D}_1 : A^{s_1} \ ;\ \mathbb{D}_2 : A^{s_2}, I\rangle \rightarrow_p \langle \mathbb{D}_1 \ ;^{s}\ \mathbb{D}_2, I \cup \{A^s \mapsto s\}\rangle}\ (\mathsf{SOLVE}_;)$$

$$\frac{\langle \mathbb{D}_1, I\rangle \rightarrow_p \langle \mathbb{D}_1', I'\rangle}{\langle \mathbb{D}_1 \ ;\ \mathbb{D}_2, I\rangle \rightarrow_p \langle \mathbb{D}_1'; \mathbb{D}_2, I'\rangle}\ (\mathsf{IND1}_;) \qquad \frac{\langle \mathbb{D}_2, I\rangle \rightarrow_p \langle \mathbb{D}_2', I'\rangle}{\langle \mathbb{D}_1 \ ;\ \mathbb{D}_2, I\rangle \rightarrow_p \langle \mathbb{D}_1; \mathbb{D}_2', I'\rangle}\ (\mathsf{IND2}_;)$$

Figure 4: Perfect relaxation on sequences.

*Solving Constraint Satisfaction Problems by Rewriting.* Constraint propagation turns a constraint satisfaction problem into an equivalent one that is easier to solve [15] by enforcing some kind of local consistency. This can be done by applying a set of *relaxation rules*. It was proved that a relaxation rule transforms a network of constraints into an equivalent one [2]. A generic *relaxation algorithm* works by applying a number of relaxation rules until no more changes can be done (in this case, we reach a *stable* network).

*Perfect Relaxation.* It has been proven [2] that if given a derivation for network of constraints that provides a proof of its construction with a context-free graph grammar (a hyper-edge replacement system), the relaxation algorithm which applies (in reverse order) the relaxation rules corresponding to the derivation, is *perfect*. In our setting, a derivation corresponds to a term, each rule can be seen as an operation (a production of the grammar), and applying them in reverse order corresponds to applying them bottom-up in the syntax tree.

In general, given a style $\langle \Sigma, \mathsf{E}, \mathsf{A}, I\rangle$, for each operator $f : S_1 \ldots S_n \rightarrow S \in \Sigma$ we can define a set of relaxation rules $\mathsf{SOLVE}_f$ as shown in Fig. 3, where for $f$ defined as $f(\mathbb{D}_1, \ldots, \mathbb{D}_n) =_{\mathsf{E}} S_{\overline{x}}[\mathbb{H}]$ we let $f^s(\mathbb{D}_1, \ldots, \mathbb{D}_n) = S^{s}_{\overline{x}}[\mathbb{H}]$ and $\widehat{f}(\mathbb{D}_1, \ldots, \mathbb{D}_n) = S_{\overline{x}}[\widehat{\mathbb{H}}]$. In words, such a rule $\mathsf{SOLVE}_f$ is applied to an $S$-sorted design whose sub-designs have been already relaxed (ensuring thus bottom-up application) and replaced by an $S^{s_i}$-labelled edge (indexed by the corresponding solution $s_i$). The effect of the rule is to change the sort $S$ to $S^s$ where $s$ is the solution of its CSP. In the computation of $s$ the use of $\widehat{f}$ ensures that the solutions $s_i$ are not re-computed.

Then, perfect relaxation can be defined by a set of rules that exploit structural induction and the set of relaxations rules. More precisely, we can define an inference system with families of rules $\mathsf{SOLVE}_f$ for operator $f$, and rule $\mathsf{IND}_f$ for exploiting induction (cf. Figure 3).

**Example 2.7.** *In our running example, relaxation can exploit the structure of the way an information filtering architecture has been designed by relying on the set of rules depicted in Figure 4. The rules inductively define perfect relaxation on sequential networks of constraints. As a simple example of how they work, consider the architecture on the left of Figure 1 which can be denoted as $\alpha; (\alpha; \alpha)$ as we saw in Example 2.2. The rules of Fig. 4 can be used to infer the sequence of relaxation steps $\langle \alpha; (\alpha; \alpha), I\rangle \rightarrow_p \langle \alpha^{s_1} \ ;\ (\alpha; \alpha), I_1\rangle \rightarrow_p \langle \alpha^{s_1} \ ;\ (\alpha^{s_1} \ ;\ \alpha), I_1\rangle \rightarrow_p \langle \alpha^{s_1} \ ;\ (\alpha^{s_1} \ ;\ \alpha^{s_1}), I_1\rangle \rightarrow_p \langle \alpha^{s_1} \ ;\ (\alpha^{s_1} \ ;^{s_2}\ \alpha^{s_1}), I_2\rangle \rightarrow_p \langle \alpha^{s_1} \ ;^{s_3}\ (\alpha^{s_1} \ ;^{s_2}\ \alpha^{s_1}), I_3\rangle$, where $I_i = I \cup \{A^{s_j} \mapsto s_j \mid 0 < j \leq i\}$ and $s_i = \{(x, y) \in \mathbb{N}^2 \mid x + i \leq y\}$. Such derivation provides a solution to the CSP of the architecture on the left of Figure 1, providing a possible assignment of security levels to its input and output ports.*

Rephrasing [2] the above mechanism provides a perfect relaxation algorithm that applies every relaxation rule only once to each occurrence of a term and the relation in the interface arc of the resulting design is the solution of the initial network of constraints. As a consequence a perfect relaxation algorithm provides a linear solution algorithm for any class of networks whose graphs are included in the language of some hyper-edge replacement system (a style in our case). Such algorithms can be understood as an application of the dynamic programming solution method (e.g. memoization may be applied). Moreover, it is not necessary to re-compute the all CSPs after a network reconfiguration: unaffected parts of the design can preserve their annotated solutions.

## 3. Conclusion

We have presented some preliminary ideas on how to provide an algebraic presentation of constraint design rewriting inspired by our previous work on ADR [4] and have exemplified it over a relaxation-based constraint solving procedure as a significant application.

We believe, that the actual design process of an architecture enriched with nonstructural constraints should involve the development of an ADR graph including both relational/numeric constraints and components/connectors representing requirement/specification/software items and their meaning. The seamless combination of the two is actually possible, and easy. For instance, graphs and constraints can be combined to conveniently represent a large variety of process algebras [16], while constraints and process algebras can be combined in the concurrent constraint programming paradigm [17, 18] to specify concurrent systems with negotiation mechanisms. The key issue is that nodes of the graphs can be considered at the same time as names (e.g., channels, links, security keys) and as variables constrained by functional and quality of service restrictions. In this spirit, we may consider design development procedures where concurrent agents act on the same constraint network design applying productions in a consistent way with consistency-preserving constraint addition operations whose effect is that of applying a refinement of the design (replacing a variable or constant in a design term by another term of the same sort). Also, we shall consider the development of constraint-guided enrichments of ADR repair and reconfiguration mechanisms [7, 19].

Constraints can also be used for requirement specification at early stages of software development. Requirements, for instance, can be expressed as ontologies enriched with hierarchical networks of constraints [20]. Requirement satisfiability, a crucial verification step at the early stages of software development, is then reduced to CSP. At run-time, instead, constraint rewrite mechanisms can be defined that recover consistency when a dynamic network of constraints becomes inconsistent due to changes in the actual constraints during execution.

## References

[1] U. Montanari, Networks of constraints: Fundamental properties and applications to picture processing, Inf. Sci. 7 (1974) 95–132.

[2] U. Montanari, F. Rossi, Constraint relaxation may be perfect, Artificial Intelligence 48 (1991) 143–170.

[3] F. Rossi, P. v. Beek, T. Walsh, Handbook of Constraint Programming (Foundations of Artificial Intelligence), Elsevier Science Inc., 2006.

[4] R. Bruni, A. Lluch Lafuente, U. Montanari, E. Tuosto, Style based architectural reconfigurations, Bulletin of the EATCS 94 (2008) 161–180.

[5] R. Bruni, A. Lluch Lafuente, U. Montanari, On structured model-driven transformations, Int. J. of Sw and Informatics 2 (1-2) (2011) 185–206.

[6] R. Bruni, A. Bucchiarone, S. Gnesi, D. Hirsch, A. Lluch Lafuente, Graph-based design and analysis of dynamic software architectures, in: P. Degano, R. De Nicola, J. Meseguer (Eds.), Concurrency, Graphs and Models, Vol. 5065 of LNCS, Springer, 2008, pp. 37–56.

[7] K. Poyias, E. Tuosto, Enforcing architectural styles in presence of unexpected distributed reconfigurations, in: M. Carbone, I. Lanese, A. Silva, A. Sokolova (Eds.), 5th Interaction and Concurrency Experience (ICE'12), Vol. 104 of EPTCS, 2012, pp. 67–82.

[8] R. Bruni, A. Lluch Lafuente, U. Montanari, Emilio Tuosto, Service Oriented Architectural Design, in: G. Barthe, C. Fournet (Eds.), 3rd International Symposium on Trustworthy Global Computing (TGC'07), Vol. 4912 of LNCS, Springer, 2007, pp. 186–203.

[9] R. Bruni, H. Foster, A. Lluch Lafuente, U. Montanari, E. Tuosto, A formal support to business and architectural design for service-oriented systems, in: Rigorous Software Engineering for Service-Oriented Systems, Vol. 6582 of LNCS, Springer, 2011, pp. 133–152.

[10] R. Bruni, F. Gadducci, A. Lluch Lafuente, An algebra of hierarchical graphs and its application to structural encoding, SACS 20 (2010) 53–96.

[11] F. Drewes, H.-J. Kreowski, A. Habel, Hyperedge replacement graph grammars, in: G. Rozenberg (Ed.), Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1, World Scientific, 1997, pp. 95–162.

[12] D. L. Métayer, Describing software architecture styles using graph grammars, IEEE Trans. Software Eng. 24 (7) (1998) 521–533.

[13] U. Montanari, F. Rossi, Perfect relaxation in constraint logic programming, in: K. Furukawa (Ed.), 8th International Conference on Logic Programming (ICLP'91), 1991, pp. 223–237.

[14] A. Corradini, U. Montanari, F. Rossi, An abstract machine for concurrent modular systems: CHARM, Theoret. Comput. Sci. 122 (1&2) (1994) 165–200.

[15] K. R. Apt, The essence of constraint propagation, Theoret. Comput. Sci. 221 (1-2) (1999) 179–210.

[16] G. L. Ferrari, D. Hirsch, I. Lanese, U. Montanari, E. Tuosto, Synchronised hyperedge replacement as a model for service oriented computing, in: F. S. de Boer, M. M. Bonsangue, S. Graf, W. P. de Roever (Eds.), 4th International Symposium on Formal Methods for Components and Objects (FMCO'05), Vol. 4111 of LNCS, 2006, pp. 22–43.

[17] V. A. Saraswat, M. C. Rinard, Concurrent constraint programming, in: F. E. Allen (Ed.), 17th ACM Symposium on Principles of Programming Languages (POPL'90), ACM Press, 1990, pp. 232–245.

[18] M. G. Buscemi, U. Montanari, Cc-pi: A constraint-based language for specifying service level agreements, in: R. De Nicola (Ed.), 16th European Symposium on Programming Languages and Systems, (ESOP'07), Vol. 4421 of LNCS, Springer, 2007, pp. 18–32.

[19] K. Poyias, E. Tuosto, A design-by-contract approach to recover the architectural style from run-time misbehaviour, Science of Computer ProgrammingTo appear.

[20] U. Montanari, E. Vassev, Soft constraints for knowlang, in: B. C. Desai, E. Vassev, S. P. Mudur, B. C. Desai (Eds.), 5th International C* Conference on Computer Science & Software Engineering (C3S2E'12), ACM, 2012, pp. 99–103.