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

We propose an algebraic approach to the design and transformation of constraint networks, inspired by *Architectural Design Rewriting*. 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. The main idea is to consider classes of constraint networks as algebras whose operators are used to denote constraint networks with terms. Constraint network transformations such as *constraint propagations* are specified with rewrite rules exploiting the network's structure provided by terms.

Keywords: Constraints, Rewriting, Hierarchical Graphs, Architectures

1. Introduction

Architectural Design Rewriting (ADR) [1] is a formal approach to the design of reconfigurable software systems that harmonizes the principles of software architectures and process calculi by means of graphical methods. ADR offers a formal setting where design development, run-time execution and reconfiguration aspects are defined on the same foot. The flexibility of ADR is witnessed by its many applications to several aspects of software engineering, including model driven transformations [2], architectural styles and reconfigurations [1], modeling of service oriented systems [3], and graphical representation of process calculi [4]. One of the main ideas of ADR is to characterize a class of graphs satisfying certain spatial constraints by means of a graph algebra.

In this paper we combine some techniques from ADR and from constraint networks [5, 6] in order to accommodate arbitrary notions of constraints, as those used in popular applications such as optimization, knowledge representation, and synchronization, to mention a few [7]. Our proposal can be understood both as (i) an enrichment of ADR with non-spatial constraints, and (ii) an application of

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

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 network's structure provided by terms.

ADR graphs can be hierarchical and, indeed, the ADR graph algebra [4] 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 [8] is employed to define an architectural style [9], 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 of representing both a graph and its syntax tree is particularly relevant for constraints networks [5, 6]. 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. The typical problem (Constraint Satisfaction Problem, CSP) is to determine all the assignments of values to variables which satisfy all constraints. These problems are clearly 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 [6], 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 of 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.

It is now clear why ADR graphs are convenient for modeling networks of constraints: not only the hierarchical structure records the steps of the design process, but also the same structure can be essential at run time for efficiently checking the satisfiability of the resulting global constraint. Also, when the more general case of Constraint Logic Programming (CLP) is considered [10], 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 graphs for supporting the design and run time evaluation of constraints is the main contribution of this paper.

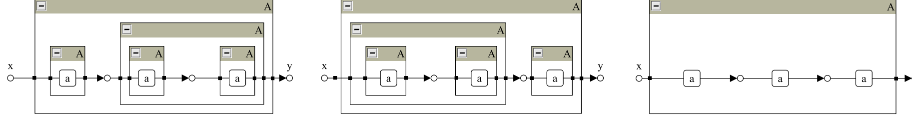

Figure 1: Two hierarchical designs (left, middle) and their flattened version (right).

2. Constraint Design Rewriting

In this section we will first present an algebraic notation for networks of constraints [6] (Section 2.1). Next (Section 2.2) we will explain how to exploit the algebraic presentation for providing an efficient mechanism for constraint solving based on rewriting techniques.

2.1. Constraint Design

Networks of constraints [6] 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 present here 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.

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 [1] to model them and the design algebra of [4] (an extension of the graph algebra of [11]) to build them.

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

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

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 algebraic reading 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 typed 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.

Terms generated by $\mathbb{G}$ and $\mathbb{D}$ are meant to represent (hierarchical) graphs and "edge-encapsulated" (hierarchical) graphs, respectively. In the following we say that a graph is *hierarchical* if it contains designs, otherwise we call it *flat* (idem for designs). The syntax has the following informal meaning: $\mathbf{0}$ represents the empty graph, x is a discrete graph containing node x only, $l\langle\overline{x}\rangle$ is a graph formed by an l-labelled hyper-edge attached to nodes $\overline{x}$ (the i-th tentacle to the i-th node in $\overline{x}$) $\mathbb{G} \mid \mathbb{H}$ is the graph resulting from the disjoint union of graphs $\mathbb{G}$ and $\mathbb{H}$ up to shared nodes, $(\nu x)\mathbb{G}$ is the graph $\mathbb{G}$ after making node x not visible

from the outside (we say that x is *restricted*), and $\mathbb{D}\langle\overline{x}\rangle$ is a graph formed by attaching design $\mathbb{D}$ to nodes $\overline{x}$ (the i-th node in the interface of $\mathbb{D}$ to the i-th node in $\overline{x}$). We let $\lfloor\overline{x}\rfloor$ denote the set of elements $\overline{x}$

A term $L_{\overline{x}}[\mathbb{G}]$ is a design labelled by L, with body graph $\mathbb{G}$ whose nodes $\overline{x}$ are exposed in the interface. A design $L_{\overline{x}}[\mathbb{G}]$ is 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. 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 $n(\cdot)$.

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 the structural graph axioms of [11] such as associativity and commutativity for $|$ with identity $\mathbf{0}$ (axioms DA1–DA3 in Definition 2.2) and name extrusion (DA4–DA6). In addition, it includes axioms to α-rename bound nodes (DA7–DA8), an axiom for making immaterial the addition of a node to a graph where that same node is already free (DA9) and another one ensuring that global names are not localised within hierarchical edges (DA10).

Definition 2.2 (design axioms). *The structural congruence $\equiv_{\mathrm{D}}$ over well-formed designs and graphs is the least congruence satisfying*

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

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

Example 2.1. *For simplicity, in this example we consider hyper-edges that have two tentacles each. Let $a \in \mathcal{E}$, $A \in \mathcal{D}$, $x, y, z, u, v, w \in \mathcal{N}$. Figure 1 depicts the designs denoted by terms $\mathbb{D}_1$ (left), $\mathbb{D}_2$ (middle) and $\mathbb{D}_3$ (right) defined below:*

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

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 is represented by a plain arrow with no head, while the second one is denoted by a normal arrow. If a node is exposed in the interface we put it on the outermost layer and overlap the edges of the various layers denoting this with black boxes on design borders.

Hierarchical designs (graphs) can be flattened into flat designs (graphs) by means of flattening rules $L_{\overline{x}}[\mathbb{G}]\langle\overline{y}\rangle \Rightarrow \mathbb{G}\{^{\overline{y}}/_{\overline{x}}\}$, which correspond to a sort of hyper-edge replacement [8]. 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 $L_{\overline{x}}[\mathbb{G}]\langle\overline{y}\rangle \Rightarrow \mathbb{G}\{^{\overline{y}}/_{\overline{x}}\} \mid L\langle\overline{y}\rangle$. 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 $\widehat{\cdot}$.

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.

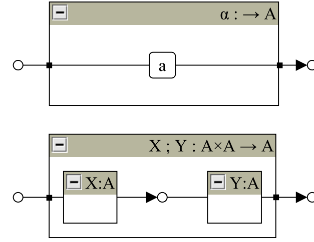

Figure 2: An algebra of sequences.

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

Example 2.2. *We define a simple style $\mathcal{S}_L = \langle \Sigma_L, \mathsf{E}_L, \mathsf{A}_L \rangle$ that characterise the set of a-labelled, acyclic, and connected sequences. The operations of the style include the constant operator $\alpha :\to A$ to introduce elements in the sequence, and a binary sequential composition $_;_ : A \times A \to A$. Both are derived operators defined by the following two equations in E_L: $\alpha = A_{u,v}[a\langle u, v\rangle]$ and $\mathbb{D}_1;\mathbb{D}_2 = A_{u,v}[(\nu w)(\mathbb{D}_1\langle u, w\rangle \mid \mathbb{D}_2\langle w, v\rangle)]$, 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, 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 A_L then A_L should also include the associativity of $_;_$ so that $\equiv_{\mathsf{A}_L}$ and $\equiv_{\mathrm{D}}$ coincide. Then, the two former terms are identified (cf. right of Figure 1 and $\mathbb{D}_3$ in Example 2.1).*

The 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.

Constraint Network Designs. Interpreting the nodes of a graph as variables and the arcs as constraints gives rise to a *network of constraints* [6]. In this work, we call a *constraint network design* a pair $\langle \mathbb{D}, I \rangle$ where $\mathbb{D}$ is a design and $I : (\mathcal{E} \cup \mathcal{D}) \to P(\mathcal{U}^k)$ 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. *Let us take the set of natural numbers $\mathbb{N}$ as the domain of nodes and take the interpretation function I that maps labels a and A of our running example to $\{(x, y) \in \mathbb{N} \mid x < y\}$. Then, the idea is that, for any design $\mathbb{D} : A$, the constraint network design $\langle \mathbb{D}, I \rangle$ constrains the nodes in $\mathbb{D}$ to be assigned a strictly ordered sequence of natural numbers.*

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 similar way as graphs are denoted by designs in ADR [4]. 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 [6] in terms of designs. Let $\langle L_{\overline{x}}[\mathbb{G}], I \rangle$ be a flat constraint network design, n abbreviate $|n(\mathbb{G})|$ and $\langle n_1, \ldots, n_n \rangle$ be any ordering of the variables (nodes) in $n(\mathbb{G})$. For the ease of presentation and without loss of generality we assume that $\mathbb{G}$ is in normal form $(\nu \overline{z})(b(\overline{y}) \mid \mathbb{H})$, with $fn(\mathbb{H}) = \emptyset$. In addition we denote the projection of a vector $\langle n_{i_1}, \ldots, n_{i_m} \rangle$ on $\overline{v} = \langle v_1, \ldots, v_n \rangle \in \mathcal{U}^n$ as $\overline{v}_{|\langle n_{i_1}, \ldots, n_{i_m} \rangle} = \langle v_{i_1}, \ldots, v_{i_m} \rangle$. The solution $sol(\langle L_{\overline{x}}[\mathbb{G}], I \rangle)$ of $\langle L_{\overline{x}}[\mathbb{G}], I \rangle$ is the set $\{\overline{v}_{|\overline{x}} \mid \overline{v} \in \mathcal{U}^n$ and $\mathbb{H} \equiv_{\mathrm{D}} b(\overline{y}) \mid \mathbb{F}$ implies $\overline{v}_{|\overline{y}} \in I(b)\}$. In other words, the CSP for a constraint network is to find the set of all the assignments of the variables connected by the interface arc such that every such assignment can be extended to an assignment of all the variables in $n(\mathbb{G})$ which satisfies all the constraints in $\mathbb{G}$, i.e. the constraints imposed by each arc $b\langle \overline{y} \rangle$. We say that a constraint network design $\langle \mathbb{D}, I \rangle$ is *consistent* if $sol(\langle \mathbb{D}, I \rangle) \neq \emptyset$. Otherwise we say that it is *inconsistent*. In general, hierarchical networks can be solved by first unfolding them. We shall see that, when the structure of the network is exploited, it is just sufficient to abstract them.

Example 2.4. *The solution of a constraint network design $\langle \mathbb{D} : A, I \rangle$ 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 α in $\mathbb{D}$, which can be computed by exploiting the structure of $\mathbb{D}$ as we shall see.*

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.

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

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

Figure 3: Perfect relaxation on generic styles.

Example 2.5. *Let us consider the style $\mathcal{S}_L$ of Example 2.3 and an interpretation function I' such that $I'(A) = I(A)$ and $I'(a) = \{(x, y) \in \mathbb{N}^2 \mid x > y\}$ 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 $I''(a) = \{(0, 1)\}$. 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 $f : S_1 \times \cdots \times S_k \rightarrow S$ 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 enjoy this property. The following style, instead is compositionally consistent.

Example 2.6. *The following style is a variant of $\mathcal{S}_L$. 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 operations of our style are parametric as well, with a families of operators $\alpha_{i,j} :\rightarrow A^{[i,j]}$ and $_;_ : A^{[i,j]} \times A^{[k,l]} \rightarrow A^{[i,l]}$ (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 $\mathbb{D}_1;\mathbb{D}_2 = A^{[i,l]}_{u,v}[(\nu w)(\mathbb{D}_1\langle u, w\rangle \mid \mathbb{D}_2\langle w, v\rangle)]$, where $\mathbb{D}_1 : A^{[i,j]}$ and $\mathbb{D}_2 : A^{[k,l]}$. The interpretation function I is such that $I(A^{[i,j]}) = I(a^{[i,j]}) = \{(x, y) \in \mathbb{N} \mid i \leq x < y \leq j\}$.*

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.

Solving Constraint Satisfaction Problems by Rewriting. Constraint propagation turns a constraint satisfaction problem into an equivalent one that is easier to solve [12] 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 [6]. 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).

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

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

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

Figure 4: Perfect relaxation on sequences.

Perfect Relaxation. It has been proven [6] 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 $\langle \Sigma, \mathsf{E}, \mathsf{A}, I\rangle$ we can define a set of relaxation rules SOLVE_f for each operator $f : S_1 \ldots S_k \rightarrow S \in \Sigma$ as defined in Fig. 3, where for f defined as $f(\mathbb{D}_1, \ldots, \mathbb{D}_n) =_{\mathsf{E}} S_{\overline{x}}[\mathbb{H}]$ we define $f^s(\mathbb{D}_1, \ldots, \mathbb{D}_n) = S^s_{\overline{x}}[\mathbb{H}]$ and $\widehat{f}(\mathbb{D}_1, \ldots, \mathbb{D}_n) = S_{\overline{x}}[\widehat{\mathbb{H}}]$. In words, such a rule 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 replace the 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 relaxations rules. More precisely, we can define an inference system with families of rules SOLVE_f rule for operator f, and rule IND_f for exploiting induction (cf. Figure 3).

Example 2.7. *Figure 4 shows the set of rules that inductively define perfect relaxation on sequential networks of constraints. Such rules can be used to infer the sequence of relaxation steps* $\langle \alpha \ ;\ \alpha, I\rangle \rightarrow_p \langle \alpha^s \ ;\ \alpha, I \cup \{A^s \mapsto s\}\rangle \rightarrow_p \langle \alpha^s \ ;\ \alpha^s, I \cup \{A^s \mapsto s\}\rangle \rightarrow_p \langle \alpha^s \ ;^{s'}\ \alpha^s, I \cup \{A^s \mapsto s, A^{s'} \mapsto s'\}\rangle$*, where* $s = \{(x, y) \in \mathbb{N}^2 \mid x + 1 \leq y\}$ *and* $s' = \{(x, y) \in \mathbb{N}^2 \mid x + 2 \leq y\}$*. Such derivation provides a solution to the CSP of* $\langle \alpha; \alpha, I\rangle$*.*

Rephrasing [6] 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 proposed an algebraic presentation of constraint design rewriting inspired by our previous work on ADR [1]. The main idea is to use (i) terms to denote (possibly hierarchical) networks of constraints and (ii) rewrite rules to denote the transformation of constraint networks. Our approach has been exemplified with relaxation-based constraint solving as a significant application.

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, graph calculi based on (synchronized) hyper-edge replacement can conveniently represent a large variety of process algebras [13], while the *cc-pi* process description language [14] combines the mobility features of π-calculus with the generality of concurrent constraint programming based on nominal soft constraints. Here the key issue is that nodes of the graphs can be considered at the same time as names in the π-calculus sense (thus representing, channels, links, continuations, security keys) and as variables required to satisfy constraints involving functional and quality of service restrictions. As future work, indeed, we may consider design development procedures in the style of concurrent constraint programming [15] where (possibly concurrent) agents act on the same constraint network design applying productions in a consistent way with *tell*-like 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).

Requirement specifications can also be expressed as ontologies enriched with hierarchical networks of constraints [16]. Checking constraint satisfiability (typically via perfect relaxation) is an important verification step at the requirement stage. 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] R. Bruni, A. Lluch Lafuente, U. Montanari, E. Tuosto, Style based architectural reconfigurations, Bulletin of the EATCS 94 (2008) 161–180.

[2] R. Bruni, A. Lluch Lafuente, U. Montanari, On structured model-driven transformations, International Journal of Software and Informatics (IJSI) 2 (1-2) (2011) 185–206.

[3] 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 Lecture Notes in Computer Science, Springer, 2011, pp. 133–152.

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

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

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

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

[8] 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.

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

[10] U. Montanari, F. Rossi, Perfect relaxation in constraint logic programming, in: ICLP, 1991, pp. 223–237.

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

[12] K. R. Apt, The essence of constraint propagation, Theoretical Computer Science 221 (1-2) (1999) 179–210.

[13] G. L. Ferrari, D. Hirsch, I. Lanese, U. Montanari, E. Tuosto, Synchronised hyperedge replacement as a model for service oriented computing, in: FMCO, Vol. 4111 of Lecture Notes in Computer Science, Springer, 2005, pp. 22–43.

[14] M. G. Buscemi, U. Montanari, Cc-pi: A constraint-based language for specifying service level agreements, in: ESOP 2007, Vol. 4421 of LNCS, Springer, 2007, pp. 18–32.

[15] V. A. Saraswat, M. C. Rinard, Concurrent constraint programming, in: F. E. Allen (Ed.), POPL, ACM Press, 1990, pp. 232–245.

[16] U. Montanari, E. Vassev, Soft constraints for knowlang, in: B. C. Desai, E. Vassev, S. P. Mudur, B. C. Desai (Eds.), C3S2E, ACM, 2012, pp. 99–103.