Science of Computer Programming 97 (2015) 23–30

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Constraint design rewriting ☆

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

[a] Dipartimento di Informatica, Università di Pisa, Italy
[b] IMT Institute for Advanced Studies, Lucca, Italy

ARTICLE INFO

Article history:
Received 27 September 2013
Accepted 5 November 2013
Available online 13 November 2013

Keywords:
Constraints
Rewriting
Hierarchical graphs
Architectures

ABSTRACT

Constraint networks are hyper-graphs whose nodes and hyper-edges respectively represent variables and relations between them. 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.

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

☆ Research supported by the European projects IP 257414 ASCENS and STReP 600708 QUANTICOL, and the Italian PRIN 2010LHT4KM CINA.
* Corresponding author.
E-mail addresses: bruni@di.unipi.it (R. Bruni), alberto.lluch@imtlucca.it (A. Lluch Lafuente), ugo@di.unipi.it (U. Montanari).

0167-6423/$ – see front matter
http://dx.doi.org/10.1016/j.scico.2013.11.015

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 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. Particularly 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 characterise 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], modelling 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 modelling networks of constraints: not only does the 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 combines Computer Science principles 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).

2. Constraint design rewriting

In this section we will first present an algebraic notation (Section 2.1) for networks of constraints [2]. Next (Section 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