

available at [www.sciencedirect.com](http://www.sciencedirect.com)journal homepage: [www.elsevier.com/locate/cosrev](http://www.elsevier.com/locate/cosrev)

# A survey of constraint-based programming paradigms<sup>☆</sup>

Maria Grazia Buscemi<sup>a</sup>, Ugo Montanari<sup>b,\*</sup>

<sup>a</sup>IMT Lucca Institute for Advanced Studies, Italy

<sup>b</sup>University of Pisa, Italy

## ARTICLE INFO

### Article history:

Received 2 October 2008

Accepted 4 October 2008

## ABSTRACT

Constraints support a programming style featuring declarative description and effective solving of several classes of problems. Unlike basic primitives of other programming languages, constraints do not specify computing operations, but rather the properties of a solution to be found. In this paper, we give a survey of the main formalisms based on constraints: Constraint Satisfaction Problems, Constraint Logic Programming and Concurrent Constraint Programming. We outline recent extensions of these approaches and we discuss ongoing trends of research.

© 2008 Elsevier Inc. All rights reserved.

## Contents

1. Introduction .....	137
2. Constraint satisfaction problems and extensions .....	138
2.1. Classical CSPs .....	138
2.2. Soft constraints .....	138
2.3. Named constraints .....	139
3. Constraint-based computational models .....	139
3.1. Constraint Logic Programming .....	139
3.2. Concurrent Constraint Programming .....	140
3.3. Concurrent Constraint Pi-calculus .....	140
4. Current trends .....	140
Acknowledgments .....	141
References .....	141

## 1. Introduction

The concept of constraint is widely used in a variety of different fields such as programming languages, artificial intelligence, databases, networks, and, more recently, computer security, web technologies and bio-informatics.

This survey outlines some of the most prominent programming paradigms relying on constraints. In the first

part of the work we recall the basic aspects of the classical Constraint Satisfaction Problem approach and we discuss two significant extensions based on semiring structures. The second part of the work is devoted to programming paradigms that embed constraints: we start by reviewing the inspiring principles of Constraint Logic Programming and Concurrent Constraint Programming and we briefly survey a line of research on combining the key features of constraint

<sup>☆</sup> Research partially supported by the EU FET IST-FP6 16004 Integrated Project SENSORIA.

\* Corresponding author.

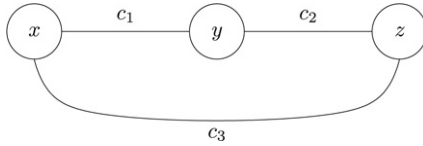
E-mail addresses: [m.buscemi@imtlucca.it](mailto:m.buscemi@imtlucca.it) (M.G. Buscemi), [ugo@di.unipi.it](mailto:ugo@di.unipi.it) (U. Montanari).



better than  $a$  and  $b$ . Moreover, 1 is the maximal and 0 the minimal element. Remarkably, several efficient algorithms defined for ordinary constraints, like constraint propagation or dynamic programming, can be generalised to  $c$ -semirings.

Typical examples of  $c$ -semiring are the  $c$ -semiring for classical CSPs ( $\{\text{False}, \text{True}\}$ , OR, AND, False, True), the  $c$ -semiring for fuzzy CSPs ( $[0, 1]$ , max, min, 0, 1), and the  $c$ -semiring for weighted CSPs ( $\mathbb{R}^+ \cup \{+\infty\}$ , min, +,  $+\infty$ , 0) called *tropical semiring* [19]. Moreover,  $c$ -semirings can be suitably composed to model more complex constraint systems.

**Example 2.** Consider the following version of the  $k$ -colorability problem shown in [Example 1](#), where we assume the  $c$ -semiring of fuzzy CSPs and we define the constraints  $c_1, c_2$  and  $c_3$  as below.



$$\begin{aligned} c_1(x \mapsto a, y \mapsto b) &= c_1(x \mapsto b, y \mapsto a) = 1 \\ c_1(x \mapsto a, y \mapsto a) &= c_1(x \mapsto b, y \mapsto b) = 0 \\ c_2(y \mapsto a, z \mapsto b) &= c_2(y \mapsto b, z \mapsto a) = 1 \\ c_2(y \mapsto a, y \mapsto a) &= c_1(x \mapsto b, y \mapsto b) = 0 \\ c_3(x \mapsto a, z \mapsto b) &= c_1(x \mapsto b, z \mapsto a) = 1 \\ c_3(x \mapsto a, z \mapsto a) &= 0.6 \\ c_3(x \mapsto b, z \mapsto b) &= 0.4. \end{aligned}$$

Intuitively,  $c_1$  and  $c_2$  simply express inequality, while  $c_3$  states that  $x \neq z$  is not a strict requirement but a preference and, in more detail, that if  $x \neq z$  is not consistent with the other constraints, the assignment  $x \mapsto a, z \mapsto a$  is preferable compared to  $x \mapsto b, z \mapsto b$ . While the classical CSP would be overconstrained if we assume just two domain values, this fuzzy version has two solutions, and the best solution corresponds to the assignment  $x \mapsto a, y \mapsto b, z \mapsto a$  that has preference value 0.6.

### 2.3. Named constraints

*Named constraints* [8] have been recently proposed as a generalisation of soft constraints that allows handling names. Named constraints rely on *named semirings*, i.e.  $c$ -semirings enriched with a permutation algebra  $A$  and a hiding operator ( $\nu \_$ ). In particular,  $A$  allows characterising the finite set of free names of each element of the named semiring  $c$  as the *support* of  $c$  in  $A$ , and  $(\nu x)c$  makes a name  $x$  local in  $c$ , in the style of process calculi. A named constraint is an element of a named semiring with an associated support.

As mentioned above, in the CSP approach constraints can be seen as functions that associate to each assignment of the variables a Boolean value. Similarly, the semiring-based constraints allow specifying extensions of the classical CSPs by associating to variable assignments values of an appropriate  $c$ -semiring. Interestingly enough, named semirings can be suitably instantiated to model either classical CSPs or generalised CSPs. In fact, the constraints  $c_1, c_2$  and  $c_3$  of [Examples 1](#) and [2](#) can be considered as values of a semiring of type  $(\{x, y, z\} \rightarrow \{a, b\}) \rightarrow S$  where  $S$  is either

the classical or the fuzzy semiring. In both cases the solution of the CSP can be expressed as  $(\nu x)(\nu y)(\nu z)c_1 \times c_2 \times c_3$ , where  $(\nu \nu)c = \sum_{\nu=a,b} c$ .

The following example shows that named constraints can also specify quite different kinds of constraints, such as those modelling Herbrand unification.

**Example 3.** A *Herbrand constraint system* is a first-order equational theory  $=_E$  satisfying the following set of rules:

$$\frac{f(t_1, \dots, t_n) =_E f(t'_1, \dots, t'_n)}{t_i =_E t'_i} \quad i = 1, \dots, n \quad \frac{x =_E t \quad t_1 =_E t_2}{[t/x]t_1 =_E [t/x]t_2}$$

with the restrictions that  $x \neq_E t(x)$  and  $f(t_1, \dots, t_n) \neq_E g(t_1, \dots, t_m)$ , where  $t(x)$  is any term different from  $x$  which contains  $x$  and  $f \neq g$ .

The named semiring  $C$  for this constraint system is as follows:

- The elements of  $C$  are all the equational theories  $E$  defined as above plus a bottom element  $\perp$ .
- $E_1 + E_2$  is the intersection of the theories  $E_1$  and  $E_2$ .
- $E_1 \times E_2$  is the unification of  $E_1$  and  $E_2$ , i.e. it is the smallest equational theory larger than or equal to  $E_1 \cup E_2$ , if it exists, otherwise  $\perp$ .
- The bottom element of  $C$  is  $\perp$  and the top element is the theory simply consisting of all equations  $t = t$ , for all terms  $t$ .
- $(\nu x)E = E \cap \bar{E}$ , where  $t_1 =_{\bar{E}} t_2$  iff  $t_1 = t_2$  or  $x$  does not occur in  $t_1, t_2$ .
- The support of a theory consists of the variables that are not bound by  $(\nu \_)$ .

For instance, given two theories  $E_1 = \{a = x\}$  and  $E_2 = \{f(x) = y\}$  where reflexive equations  $t = t$  are omitted,  $E_1 \times E_2$  is the unification of  $E_1$  and  $E_2$ , i.e.  $E_1 \times E_2 = \{a = x, f(x) = y, f(x) = f(a), f(a) = y\}$ .

## 3. Constraint-based computational models

### 3.1. Constraint Logic Programming

Constraint Logic Programming (CLP) has been introduced by Jaffar and Lassez [12,15,1] as an extension to Logic Programming (LP). CLP shares the declarative flavour of LP, according to which the programmer specifies *what* to compute while disregarding how to compute a program. The extension amounts to considering an arbitrary domain of data rather than the standard Herbrand universe of LP, and by replacing the notion of unification with the concept of constraint solving over such a domain. Consequently, the unifiability test becomes a test of consistency of the new constraint with respect to the constraint store, namely the set of constraints previously accumulated.

A CLP program consists of a finite set of rules whose bodies contain conjunctions of literals, i.e. ordinary atomic goals with no function symbols, and constraints over a certain domain. While the declarative semantics is the same as for LP, the operational semantics features a constraint-solver for the constraints and an adaptation of the goal-reduction technique of LP. In more detail, the operational semantics is

given in terms of derivations (i.e. reductions) from the goal. At each reduction step, the leftmost literal of the goal is rewritten. Assuming the literal is a primitive constraint, if it is consistent with the current store, then it is added, otherwise the derivation fails. Conversely, if the literal is an atom, it is reduced using one of the rules of its definition.

**Example 4.** Consider the following simple CLP program that defines the predicate  $\text{min}(x, y, z)$  such that  $z = \min\{x, y\}$ .

$$\begin{aligned} \text{min}(X, Y, Z) &: - X \leq Y, Z = X \\ \text{min}(X, Y, Z) &: - Y \leq X, Z = Y. \end{aligned}$$

$X \leq Y$  and  $Y \leq X$  are primitive constraints that, along with standard LP atoms  $\text{min}(X, Y, Z)$  are regarded as literals. For instance, the execution of the goal  $\text{min}(2, 4, W)$  can produce the answer  $W = 2$ , while the execution  $\text{min}(1, 2, 3)$  fails.

In [3], CLP has been extended to handle soft constraints. In the resulting paradigm, which is called Soft Constraint Logic Programming (SCLP), a program is essentially a CLP program (with Herbrand function symbols) where clause bodies can be semiring values. Clause satisfaction is defined in terms of semiring entailment (i.e. the semiring partial order  $\leq$ ).

### 3.2. Concurrent Constraint Programming

Concurrent Constraint Programming (CCP) is a simple and powerful computing paradigm introduced by Saraswat [21, 22] that relies on a very general, formal notion of constraint system. The main difference with respect to imperative programming languages concerns the notion of store, which represents the state of a system. In CCP, rather than containing variable instantiations, the store is a constraint that specifies *partial information* about the possible values the variables can take at any stage of the computation. Moreover, CCP extends CLP by allowing concurrent processes that share a common store representing the constraint established until that moment. Processes can interact with each other by adding a constraint if it is consistent with the store ( $\text{tell}$  action). Alternatively, a process can check if the store *entails* (implies) a given constraint ( $\text{ask}$  action) and, if this is not the case, it remains blocked until some concurrent process adds enough information to the store. Hence, as computation proceeds, more and more information is accumulated and the store is *monotonically refined*.

The concepts of entailment and consistency are formulated according to Dana Scott's information systems approach to domain theory [25]. A constraint system consists of a set of basic constraints, called *tokens*, and an entailment relation which specifies, at each state of the computation, if a certain token is entailed by a given set of tokens. CCP supports variable hiding in constraints by relying on Tarski's cylindric algebras [11]. A cylindric constraint system is obtained from a constraint system by adding a set of variables and a family of functions  $\exists_x$ , for every variable  $x$ , that basically captures the projection of information induced by existential quantifiers.

**Example 5.** Assume a simple constraint system based on first-order theories with equality. Consider the following CCP processes.

$$\begin{aligned} A &= \text{tell}(x = a) \rightarrow \text{ask}(y = f(a)) \rightarrow \text{Success} \\ B &= \text{tell}(y = f(x)) \rightarrow \text{Success} \\ C &= \exists_x \text{tell}(y = f(x)) \rightarrow \text{Success}. \end{aligned}$$

The process  $A$ , after the first  $\text{tell}$  action, is stuck since  $y = f(a)$  is not (logically) entailed by  $x = a$ . Instead, the parallel composition of  $A$  and  $B$  can successfully terminate. Indeed, after the execution of the first step of  $A$  and the execution of  $B$ , the store is  $x = a \wedge y = f(x)$ , which entails  $y = f(a)$ . Conversely, the parallel composition of  $A$  and  $C$  cannot successfully terminate since after two steps the store  $x = a \wedge \exists_x(y = f(x))$  does not entail  $y = f(a)$ .

Several extensions of the pure CCP paradigm have been proposed. In timed CCP [23,18], processes cannot wait indefinitely for an event and, in case a time-out occurs, they must take an alternative action. The Soft CCP model [5] generalises CCP to handle soft constraints: the novel idea is to parametrise  $\text{tell}$  and  $\text{ask}$  primitives with a preference level that is used to determine their success, failure or suspension.

### 3.3. Concurrent Constraint Pi-calculus

A number of contributions in the literature (see, e.g., [26]) focus on merging concurrency and problem solving aspects. Recently, some efforts have been made to enrich nominal process calculi like the pi-calculus [16] with primitives for constraint handling. The concurrent constraint pi-calculus *cc-pi calculus* [8] features a symmetric, synchronous mechanism of interaction between senders and receivers, where the sent name is 'fused' (i.e. identified) to the received name, and such an *explicit fusion* allows using interchangeably the two names. Moreover, the *cc-pi calculus* generalises explicit fusions to be constraints over an arbitrary named semiring and introduces primitives for creating, removing and making logical checks on constraints. The *cc-pi calculus* also includes a restriction operation *à la pi-calculus* that allows for local stores of constraints. Synchronisations may have the effect of combining local stores of interacting processes into a global store.

**Example 6.** Consider the following two processes

$$P = x = v \mid \bar{x}(z).P' \quad \text{and} \quad Q = v = y \mid y(w).Q'$$

with  $\bar{x}(z)$  an output action,  $y(w)$  an input action and  $x = v$  and  $v = y$  two constraints.  $P$  and  $Q$  can synchronise because the identification of  $x$  and  $y$  is entailed by the combination of constraints in parallel  $x = v \times v = y$  and because the name fusion  $z = w$  is consistent with the store  $x = v \times v = y$ . The system resulting from such synchronisation is  $P' \mid Q' \mid x = v \times v = y \times z = w$ .

## 4. Current trends

Many research results are being published on using constraint-based approaches in different application fields. We conclude this survey by mentioning some new contributions in the area of Service Oriented Computing (SOC). We refer to the bibliography and, in particular, to [20] for more recent applications to other areas, such as security and bio-informatics.

Service Oriented Computing is an emerging paradigm that builds upon the notion of services as interoperable

elements that can be described, published, searched and composed. Services may expose both functional properties (i.e. what they do) and non-functional properties (i.e. the way they are supplied). A Service Level Agreement (SLA) is a contract between two parties, usually a service provider and a customer, that records non-functional properties about a service like performance, availability, and cost. Constraint-based paradigms are showing to be quite promising in specifying and verifying SLAs. For instance, the above-mentioned cc-pi calculus has been conceived as a simple model of contracts for SLAs that also allows studying mechanisms for joining different SLA requirements [7]. The idea is to specify each SLA parameter as a variable and each SLA requirement or guarantee as a constraint that connects the involved variables. The parties are modelled as communicating processes. A constraint can be generated either by a single process or by the synchronisation of two processes that induces the identification of the communicated values. Remarkably, such a method can not only specify SLA negotiations, but also run-time check that a SLA is not violated by the involved parties.

Another constraint-based model for SLAs is presented in [6]. The proposed framework extends SCCP to allow for non-monotonic evolution of the constraint store. Specifically, the authors introduce operations for removing constraints from a store, for relaxing the constraints involving a given set of variables and then adding a new constraint, and for checking if a constraint is not entailed by the store.

---

## Acknowledgments

The authors wish to thank Stefano Bistarelli and Francesca Rossi for helpful comments.

## REFERENCES

- [1] K.R. Apt, M. Wallace, *Constraint Logic Programming using Eclipse*, Cambridge University Press, 2007.
- [2] S. Bistarelli, *Semirings for Soft Constraint Solving and Programming*, Springer, 2004.
- [3] S. Bistarelli, U. Montanari, F. Rossi, Semiring-based constraint logic programming, in: *Proc. IJCAI'97*, 1997.
- [4] S. Bistarelli, U. Montanari, F. Rossi, Semiring-based constraint satisfaction and optimization, *Journal of the ACM* 44 (2) (1997) 201–236.
- [5] S. Bistarelli, U. Montanari, F. Rossi, Soft concurrent constraint programming, *ACM Transactions on Computational Logic* 7 (3) (2006) 563–589.
- [6] S. Bistarelli, F. Santini, A nonmonotonic soft concurrent constraint language for sla negotiation, in: *Proc. CILC'08*, 2008.
- [7] M.G. Buscemi, L. Ferrari, C. Moiso, U. Montanari, Constraint-based policy negotiation and enforcement for telco services, in: *Proc. TASE'07*, IEEE Computer Society, 2007, pp. 463–472.
- [8] M.G. Buscemi, U. Montanari, Cc-pi: A constraint-based language for specifying service level agreements, in: *Proc. ESOP'07*, in: *Lecture Notes in Computer Science*, vol. 4421, Springer, 2007, pp. 18–32.
- [9] R. Dechter, *Constraint Processing*, Morgan Kaufmann Publishers, 2003.
- [10] D. Dubois, H. Fargier, H. Prade, Using fuzzy constraints in job-shop scheduling, in: *Proc. IJCAI'93/SIGMAN Workshop on Knowledge-based on Production Planning, Scheduling and Control*, 1993.
- [11] L. Henkin, A. Tarski, J.D. Monk, *Cylindric Algebras*, North-Holland, Amsterdam, 1971.
- [12] J. Jaffar, J.-L. Lassez, Constraint logic programming, in: *Proc. POPL'87*, ACM Press, 1987.
- [13] G. Kondrak, P. van Beek, A theoretical evaluation of selected backtracking algorithms, *Artificial Intelligence* 89 (1997) 365–387.
- [14] A.K. Mackworth, Consistency in networks of relations, *Artificial Intelligence* 8 (1977) 99–118.
- [15] K. Marriott, P.J. Stuckey, *Introduction to Constraint Logic Programming*, MIT Press, 1998.
- [16] R. Milner, J. Parrow, D. Walker, A calculus of mobile processes, i and ii, *Information and Computation* 100 (1) (1992) 1–40.
- [17] U. Montanari, Networks of constraints: Fundamental properties and application to picture processing, *Information Science* 7 (1974) 95–132.
- [18] C. Palamidessi, F.D. Valencia, A temporal concurrent constraint programming calculus, in: *Proc. CP'01*, Springer, 2001, pp. 302–316.
- [19] J.E. Pin, *Tropical semirings*, in: *Idempotency*, Cambridge University Press, 1998, pp. 50–69.
- [20] F. Rossi, P. van Beek, T. Walsh, *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*, Elsevier Science, 2006.
- [21] V. Saraswat, *Concurrent Constraint Programming*, MIT Press, 1993.
- [22] V. Saraswat, M. Rinard, Concurrent constraint programming, in: *Proc. POPL'90*, ACM Press, 1990.
- [23] V.A. Saraswat, R. Jagadeesan, V. Gupta, Timed default concurrent constraint programming, *Journal of Symbolic Computation* 22 (5–6) (1996) 475–520.
- [24] T. Schiex, H. Fargier, G. Verfaillie, Valued constraint satisfaction problems: Hard and easy problems, in: *Proc. IJCAI'95*, 1995.
- [25] D.S. Scott, Domains for denotational semantics, in: *Proc. ICALP'82*, in: *Lecture Notes in Computer Science*, vol. 140, Springer, 1982, pp. 577–613.
- [26] G. Smolka, The Oz programming model, in: *Computer Science Today*, in: *Lecture Notes in Computer Science*, vol. 1000, Springer, 1995, pp. 324–343.