A survey of constraint-based programming paradigms

Maria Grazia Buscemi\textsuperscript{a}, Ugo Montanari\textsuperscript{b,∗}

\textsuperscript{a} IMT Lucca Institute for Advanced Studies, Italy
\textsuperscript{b} University of Pisa, Italy

\section*{Article Info}

Article history:
Received 2 October 2008
Accepted 4 October 2008

Contents

1. Introduction .................................................................................................................. 1
2. Constraint satisfaction problems and extensions ......................................................... 2
2.1. Classical CSPs .......................................................................................................... 2
2.2. Soft constraints ......................................................................................................... 2
2.3. Named constraints ................................................................................................. 3
3. Constraint-based computational models ....................................................................... 3
3.1. Constraint Logic Programming (CLP) .................................................................. 3
3.2. Concurrent Constraint Programming ................................................................... 4
3.3. Concurrent Constraint Pi-calculus ....................................................................... 4
4. Current trends .............................................................................................................. 4
Acknowledgments ......................................................................................................... 5
References ....................................................................................................................... 5

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 programming with process calculi. We conclude with some current trends of research.

We do not claim this survey to be exhaustive in any way. Rather, our work is conceived as a review from a personal perspective of selected classical and more recent contributions in the field of constraint-based computational programming.
models. We refer the interested reader to the bibliography and, specifically, to [20] for a comprehensive treatment of most of the topics addressed in this work.

2. Constraint satisfaction problems and extensions

2.1. Classical CSPs

The classical Constraint Satisfaction Problem (CSP) framework [17,14] is a well-known paradigm, that is suited to specify many kinds of real-life problems and that has been broadly investigated in computer science and artificial intelligence. The key idea underlying CSP is to solve a problem by stating constraints representing requirements about the problem and, then, finding solutions satisfying all the constraints.

A CSP defined over a constraint network consists of a finite set of variables, each associated with a domain of values and a set of constraints. A constraint, or even a network of constraints, states the legal combinations of values of a certain subset of variables. Formally, constraints are functions which, given an assignment of the variables to some domain, return a Boolean value. In this sense, constraints are declarative, namely they specify which relationship must hold while disregarding the computational procedure to enforce that relationship. A solution to a CSP is an assignment to each variable of a value from its domain such that all the constraints are satisfied. A CSP can be represented as a constraint graph, whose nodes correspond to the variables of the problem and whose (hyper)arcs correspond to its constraints.

Example 1. Consider the following CSP consisting of a system of three variables that can be instantiated to k domain values such that pairs of adjacent variables (i.e., variables connected by a constraint) hold distinct values.

```
x \neq y
y \neq z
z \neq x
```

The CSP above is an instance of the well-known k-colorability problem when associating the nodes of the graph with the variables, the set of possible colors with the domain and the not-equal constraints between adjacent nodes with the constraints of the problem. Of course, if in our example we take k \leq 2 the problem is overconstrained, i.e. it has no solution.

A prominent line of research within CSPs focuses on developing algorithms for constraint propagation and solving. Constraint propagation (see, e.g., [17,14]) roughly amounts to choosing a small subset of a constraint net and to making some of its local constraints stronger by forbidding some value assignment which would make the subnet overconstrained. By repeating the process for several overlapping subnets, global information is propagated at the local level, until no additional modification is possible. Then backtracking search [13] must be applied. A backtracking search algorithm builds a partial solution by assigning values to variables one by one; if it finds a dead end in which the partial solution cannot be consistently extended, it undoes a choice it made and tries another one; this process is iterated until all possibilities are tried or until a solution is found. Backtracking search algorithms are complete in the sense that they eventually find a solution if one exists. CSPs are in general NP-complete. However, much work has been done on identifying restrictions on CSPs such that solving can be done efficiently. There are two main restrictions leading to tractability: restricting the topology of the underlying graph and restricting the type of the allowed constraints. A significant example of methods that are well suited in the presence of the former kind of restriction is dynamic programming. Dynamic programming techniques [9] can be efficiently applied to certain constraint networks having a “thick” tree structure and they allow solving a CSP by solving some of its subproblems and then by combining the results to obtain a solution of the whole problem.

2.2. Soft constraints

Classical CSPs are not suited in several real-life scenarios. Indeed, CSPs are not able to model constraints that are preferences rather than strict requirements or to provide a not-so-bad solution when the problem is overconstrained.

A large amount of work in the literature focuses on generalisations of classical constraints to soft constraints.

Roughly, a soft constraint is a constraint that rather than returning a Boolean yields more informative values such as a preference value or a cost. When combining constraints, one has to take into account such additional values. Hence, it is necessary to have an underlying formalism that provides suitable operations for combining and comparing constraints.

Some extensions of classical CSPs give specific interpretations of soft constraints like weighted CSPs for modelling cost functions, probabilistic CSPs or fuzzy CSPs. Fuzzy constraints [10], for instance, generalise classical constraints to membership functions, which roughly express the fact that a value assignment to a set of variables is partially permitted. On the other side, semiring-based constraints [2,4] and valued constraints [24] are more generic extensions to soft constraints, in the sense that they can model different kinds of constraints by varying their underlying structure.

A constraint semiring (c-semiring) [4] is an algebra ([A, +, \times], 0, 1), where (+, 0) and (\times, 1) are commutative monoids, + is idempotent, \times distributes over +, 1 and 0 are absorbing elements for + and \times respectively (i.e., a + 1 = 1 and a \times 0 = 0 for all a in A). C-semirings are also equipped with a partial ordering \leq such that a \leq b iff a + b = b, which means that a is worse than b, or, more interestingly, that a entails b. Intuitively, the preference level associated to each variable instantiation is modelled as a value of a c-semiring; the combination of constraints is expressed by the product operation, while the sum operation takes care of combining constraints.

Typical examples of c-semirings are the c-semirings for classical CSPs ([False, True], OR, AND, False, True), the
c-semiring for fuzzy CSPs \([0, 1]\), max, min, 0, 1), and the c-semiring for weighted CSPs \([\mathbb{R}^+ \cup \{-\infty\}, \min, +, +, +, 0, 0]\) called tropical semirings [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 c1, c2 and c3 as below.

\[
\begin{align*}
\text{c}_1(\text{x} \mapsto a, \text{y} \mapsto b) &= \text{c}_1(\text{x} \mapsto b, \text{y} \mapsto a) = 1 \\
\text{c}_2(\text{x} \mapsto a, \text{y} \mapsto a) &= \text{c}_1(\text{x} \mapsto b, \text{y} \mapsto b) = 0 \\
\text{c}_2(\text{y} \mapsto a, \text{z} \mapsto b) &= \text{c}_2(\text{y} \mapsto b, \text{z} \mapsto a) = 1 \\
\text{c}_2(\text{y} \mapsto a, \text{z} \mapsto a) &= \text{c}_1(\text{x} \mapsto b, \text{z} \mapsto b) = 0 \\
\text{c}_3(\text{x} \mapsto a, \text{z} \mapsto b) &= \text{c}_1(\text{x} \mapsto b, \text{z} \mapsto a) = 0.6 \\
\text{c}_3(\text{x} \mapsto b, \text{z} \mapsto b) &= 0.4.
\end{align*}
\]

Intuitively, c1 and c2 simply express inequality, while c3 states that x ≠ z is not a strict requirement but a preference and, in more detail, that if x ≠ z is not consistent with the other constraints, the assignment x ↦ a, z ↦ a is preferable compared to x ↦ b, z ↦ 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 ↦ a, y ↦ b, z ↦ 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 (\(\oplus\)). 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 \((x \land c)\) 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 c1, c2 and c3 of Examples 1 and 2 can be considered as values of a semiring of type \((\mathbb{X} \times \mathbb{Y} \times \mathbb{Z}) \mapsto (\mathbb{A}, \mathbb{B}) \mapsto S\) where S is either the classical or the fuzzy semiring. In both cases the solution of the CSP can be expressed as \((x^t, y^t, z^t) = \sum_{a \in \mathbb{A}, b \in \mathbb{B}} c_{x \mapsto a} \times c_{y \mapsto b} \times c_{z \mapsto c}\), where \((x \mapsto a, y \mapsto b) = \sum_{a \in \mathbb{A}, b \in \mathbb{B}} c_{x \mapsto a} \times c_{y \mapsto b} \times c_{z \mapsto c}\).

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 \(\Gamma\) satisfying the following set of rules:

\[
f(t_1, \ldots, t_n) = f(t'_1, \ldots, t'_{m})
\]

\[
t_i \Gamma \rightarrow t'_i \Gamma
\]

\[
x = t \Gamma \rightarrow t \Gamma
\]

\[
[t/x]t_1 \Gamma \rightarrow [t/x]t_2 \Gamma
\]

with the restrictions that \(x \neq t\) and \(f(t_1, \ldots, t_n) \neq g(t_1, \ldots, t_m)\), 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 equations \(\Gamma\) defined above plus a bottom element \(-\infty\).
- \(E_1 + E_2\) is the intersection of the theories \(E_1\) and \(E_2\).
- \(E_1 \times E_2\) is the union 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 \(-\infty\).
- The bottom element of C is \(-\infty\) and the top element is the theory simply consisting of all equations \(t = t\) for all terms t.
- \((v \times x) = E \cap \tilde{x}\) where \(t_1 =_E t_2\) 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 \((v \times x)\).

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 (CLP)

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 \(z = \text{min}(x, y)\).

\[
x =_{\text{min}} t \Gamma \rightarrow t \Gamma
\]

\[
[t/x]t_1 \Gamma \rightarrow [t/x]t_2 \Gamma
\]

Assume a simple constraint system based on the store is a composition of partial information about a customer, that records non-functional properties about a service. 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 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 parameterise tell and 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

\[ P = x = u \mid \pi(x), P' \quad \text{and} \quad Q = v = y \mid \gamma(w), Q' \]

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

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

\[ \min(X, Y, Z) : = -X \leq Y, Z = X \]

\[ \min(X, Y, Z) : = -Y \leq X, Z = Y. \]

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

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 ≤).

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 (tell action). Alternatively, a process can check if the store entails (implies) a given constraint (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.

\[ A = \text{tell}(x = a) \rightarrow \text{ask}(y = f(a)) \rightarrow \text{Success} \]

\[ B = \text{tell}(y = f(x)) \rightarrow \text{Success} \]

\[ C = \exists_x \text{tell}(y = f(x)) \rightarrow \text{Success}. \]

The process \( A \), after the first 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 \( x = a \land y = f(x) \), which entails \( y = f(a) \). Conversely, the parallel composition of \( A \) and \( C \) cannot successfully terminate since after two steps the store \( x = a \land \exists_y(y = f(x)) \) does not entail \( y = f(a) \).
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