

# Soft Concurrent Constraint Programming

Stefano Bistarelli

Istituto di Informatica e Telematica, C.N.R., Pisa, Italy

and

Ugo Montanari

Dipartimento di Informatica, Università di Pisa, Italy

and

Francesca Rossi

Dipartimento di Matematica Pura ed Applicata, Università di Padova, Italy.

---

Soft constraints extend classical constraints to represent multiple consistency levels, and thus provide a way to express preferences, fuzziness, and uncertainty. While there are many soft constraint solving formalisms, even distributed ones, by now there seems to be no concurrent programming framework where soft constraints can be handled. In this paper we show how the classical concurrent constraint (cc) programming framework can work with soft constraints, and we also propose an extension of cc languages which can use soft constraints to prune and direct the search for a solution. We believe that this new programming paradigm, called soft cc (scc), can be also very useful in many web-related scenarios. In fact, the language level allows web agents to express their interaction and negotiation protocols, and also to post their requests in terms of preferences, and the underlying soft constraint solver can find an agreement among the agents even if their requests are incompatible.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming—*Distributed programming*; D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*Semantics*; *Syntax*; D.3.2 [**Programming Languages**]: Language Classifications—*Concurrent, distributed, and parallel languages*; *Constraint and logic languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Concurrent programming structures*; *Constraints*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Operational semantics*

General Terms: Languages

Additional Key Words and Phrases: constraints, soft constraints, concurrent constraint programming

---

## 1. INTRODUCTION

The concurrent constraint (cc) paradigm [Saraswat 1993] is a very interesting computational framework which merges together constraint solving and concurrency. The main idea is to choose a *constraint system* and use constraints to model com-

---

Research supported in part by the the the Italian MIUR Projects COMETA and NAPOLI and by ASI project ARISCOM.

Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.

© 2018 ACM 1529-3785/2018/0700-0001 \$5.00

munication and synchronization among concurrent agents.

Until now, constraints in cc were *crisp*, in the sense that they could only be satisfied or violated. Recently, the classical idea of *crisp* constraints has been shown to be too weak to represent real problems and a big effort has been done toward the use of soft constraints [Freuder and Wallace 1992; Dubois et al. 1993; Ruttkay 1994; Fargier and Lang 1993; Schiex et al. 1995; Bistarelli et al. 1997; 2001; Bistarelli 2001], which can have more than one level of consistency. Many real-life situations are, in fact, easily described via constraints able to state the necessary requirements of the problems. However, usually such requirements are not hard, and could be more faithfully represented as preferences, which should preferably be satisfied but not necessarily. Also, in real life, we are often challenged with over-constrained problems, which do not have any solution, and this also leads to the use of preferences or in general of soft constraints rather than classical constraints.

Generally speaking, a soft constraint is just a classical constraint plus a way to associate, either to the entire constraint or to each assignment of its variables, a certain element, which is usually interpreted as a level of preference or importance. Such levels are usually ordered, and the order reflects the idea that some levels are better than others. Moreover, one has also to say, via suitable combination operators, how to obtain the level of preference of a global solution from the preferences in the constraints.

Many formalisms have been developed to describe one or more classes of soft constraints. For instance consider fuzzy CSPs [Dubois et al. 1993; Ruttkay 1994], where crisp constraints are extended with a level of preference represented by a real number between 0 and 1, or probabilistic CSPs [Fargier and Lang 1993], where the probability to be in the real problem is assigned to each constraint. Some other examples are partial [Freuder and Wallace 1992] or valued CSPs [Schiex et al. 1995], where a preference is assigned to each constraint, in order to satisfy as many constraints as possible, and thus handle also overconstrained problems.

We think that many network-related problem could be represented and solved by using soft constraints. Moreover, the possibility to use a concurrent language on top of a soft constraint system, could lead to the birth of new protocols with an embedded constraint satisfaction and optimization framework.

In particular, the constraints could be related to a quantity to be minimized/maximized but they could also satisfy policy requirements given for performance or administrative reasons. This leads to change the idea of QoS in routing and to speak of *constraint-based* routing [Awduche et al. 1999; Clark 1989; Jain and Sun 2000; Calisti and Faltings 2000]. Constraints are in fact able to represent in a declarative fashion the needs and the requirements of agents interacting over the web.

The features of soft constraints could also be useful in representing routing problems where an imprecise state information is given [Chen and Nahrstedt 1998]. Moreover, since QoS is only a specific application of a more general notion of Service Level Agreement (SLA), many applications could be enhanced by using such a framework. As an example consider E-commerce: here we are always looking for establishing an agreement between a merchant, a client and possibly a bank. Also, all auction-based transactions need an agreement protocol. Moreover, also secu-

ality protocol analysis have shown to be enhanced by using security levels instead of a simple notion of secure/insecure level [Bella and Bistarelli 2001]. All these considerations advocate for the need of a soft constraint framework where optimal answers are extracted.

In this paper, we use one of the frameworks able to deal with soft constraints [Bistarelli et al. 1995; 1997]. The framework is based on a semiring structure that is equipped with the operations needed to combine the constraints present in the problem and to choose the best solutions. According to the choice of the semiring, this framework is able to model all the specific soft constraint notions mentioned above. We compare the semiring-based framework with constraint systems “*a la Saraswat*” and then we show how use it inside the cc framework. The next step is the extension of the syntax and operational semantics of the language to deal with the semiring levels. Here, the main novelty with respect to cc is that tell and ask agents are equipped with a preference (or consistency) threshold which is used to prune the search.

After a short summary of concurrent constraint programming (§2.1) and of semiring-based SCSPs (§2.2), we show how the concurrent constraint framework can be used to handle also soft constraints (§3). Then we integrate semirings inside the syntax of the language and we change its semantics to deal with soft levels (§4). Some notions of observables able to deal with a notion of optimization and with *success* (§6.1), *fail* (§6.2) and *hang* computations (§6.3) are then defined. Some examples (§5) and an application scenario (§7) conclude our presentation showing the expressivity of the new language. Finally, conclusions (§8) are added to point out the main results and possible directions for future work.

## 2. BACKGROUND

### 2.1 Concurrent Constraint Programming

The concurrent constraint (cc) programming paradigm [Saraswat 1993] concerns the behaviour of a set of concurrent agents with a shared store, which is a conjunction of constraints. Each computation step possibly adds new constraints to the store. Thus information is monotonically added to the store until all agents have evolved. The final store is a refinement of the initial one and it is the result of the computation. The concurrent agents do not communicate directly with each other, but only through the shared store, by either checking if it entails a given constraint (*ask* operation) or adding a new constraint to it (*tell* operation).

**2.1.1 Constraint Systems.** A constraint is a relation among a specified set of variables. That is, a constraint gives some information on the set of possible values that these variables may assume. Such information is usually not complete since a constraint may be satisfied by several assignments of values of the variables (in contrast to the situation that we have when we consider a valuation, which tells us the only possible assignment for a variable). Therefore it is natural to describe constraint systems as systems of *partial* information [Saraswat 1993].

The basic ingredients of a constraint system (defined following the information systems idea) are a set  $D$  of *primitive constraints* or *tokens*, each expressing some partial information, and an entailment relation  $\vdash$  defined on  $\wp(D) \times D$  (or its

extension defined on  $\wp(D) \times \wp(D)$ <sup>1</sup> satisfying:

- $u \vdash P$  for all  $P \in u$  (reflexivity) and
- if  $u \vdash v$  and  $v \vdash z$ , then  $u \vdash z$  (transitivity).

We also define  $u \approx v$  if  $u \vdash v$  and  $v \vdash u$ .

As an example of entailment relation, consider  $D$  as the set of equations over the integers; then  $\vdash$  could include the pair  $\langle \{x = 3, x = y\}, y = 3 \rangle$ , which means that the constraint  $y = 3$  is entailed by the constraints  $x = 3$  and  $x = y$ . Given  $X \in \wp(D)$ , let  $\overline{X}$  be the set  $X$  closed under entailment. Then, a constraint in an information system  $\langle \wp(D), \vdash \rangle$  is simply an element of  $\overline{\wp(D)}$ .

As it is well known,  $\langle \overline{\wp(D)}, \subseteq \rangle$  is a complete algebraic lattice, the compactness of  $\vdash$  gives the algebraic structure for  $\overline{\wp(D)}$ , with least element  $true = \{P \mid \emptyset \vdash P\}$ , greatest element  $D$  (which we will mnemonically denote *false*), glbs (denoted by  $\sqcap$ ) given by the closure of the intersection and lubs (denoted by  $\sqcup$ ) given by the closure of the union. The lub of chains is, however, just the union of the members in the chain. We use  $a, b, c, d$  and  $e$  to stand for elements of  $\overline{\wp(D)}$ ;  $c \subseteq d$  means  $c \vdash d$ .

**2.1.2 The hiding operator: Cylindric Algebras.** In order to treat the hiding operator of the language (see later), a general notion of existential quantifier for variables in constraints is introduced, which is formalized in terms of cylindric algebras. This leads to the concept of *cylindric constraint system* over an infinite set of variables  $V$  such that for each variable  $x \in V$ ,  $\exists_x : \overline{\wp(D)} \rightarrow \overline{\wp(D)}$  is an operation satisfying:

- (1)  $u \vdash \exists_x u$ ;
- (2)  $u \vdash v$  implies  $(\exists_x u) \vdash (\exists_x v)$ ;
- (3)  $\exists_x(u \sqcup \exists_x v) \approx (\exists_x u) \sqcup (\exists_x v)$ ;
- (4)  $\exists_x \exists_y u \approx \exists_y \exists_x u$ .

**2.1.3 Procedure calls.** In order to model parameter passing, *diagonal elements* are added to the primitive constraints. We assume that, for  $x, y$  ranging in  $V$ ,  $\overline{\wp(D)}$  contains a constraint  $d_{xy}$  which satisfies the following axioms:

- (1)  $d_{xx} = true$ ,
- (2) if  $z \neq x, y$  then  $d_{xy} = \exists_z(d_{xz} \sqcup d_{zy})$ ,
- (3) if  $x \neq y$  then  $d_{xy} \sqcup \exists_x(c \sqcup d_{xy}) \vdash c$ .

Note that the in the previous definition we assume the cardinality of the domain for  $x, y$  and  $z$  greater than 1. Note also that, if  $\vdash$  models the equality theory, then the elements  $d_{xy}$  can be thought of as the formulas  $x = y$ .

**2.1.4 The language.** The syntax of a cc program is show in Table I:  $P$  is the class of programs,  $F$  is the class of sequences of procedure declarations (or clauses),  $A$  is the class of agents,  $c$  ranges over constraints, and  $x$  is a tuple of variables. Each procedure is defined (at most) once, thus nondeterminism is expressed via the  $+$  combinator only. We also assume that, in  $p(x) :: A$ , we have  $vars(A) \subseteq x$ , where

<sup>1</sup>The extension is s.t.  $u \vdash v$  iff  $u \vdash P$  for every  $P \in v$ .

Table I. cc syntax

---

$P ::= F.A$
$F ::= p(x) :: A \mid F.F$
$A ::= \text{success} \mid \text{fail} \mid \text{tell}(c) \rightarrow A \mid E \mid A \parallel A \mid \exists_x A \mid p(x)$
$E ::= \text{ask}(c) \rightarrow A \mid E + E$

---

$\text{vars}(A)$  is the set of all variables occurring free in agent  $A$ . In a program  $P = F.A$ ,  $A$  is the initial agent, to be executed in the context of the set of declarations  $F$ . This corresponds to the language considered in [Saraswat 1993], which allows only guarded nondeterminism.

In order to better understand the extension of the language that we will introduce later, let us remind here the operational semantics of the agents.

- agent “*success*” succeeds in one step,
- agent “*fail*” fails in one step,
- agent “ $\text{ask}(c) \rightarrow A$ ” checks whether constraint  $c$  is entailed by the current store and then, if so, behaves like agent  $A$ . If  $c$  is inconsistent with the current store, it fails, and otherwise it suspends, until  $c$  is either entailed by the current store or is inconsistent with it;
- agent “ $\text{ask}(c_1) \rightarrow A_1 + \text{ask}(c_2) \rightarrow A_2$ ” may behave either like  $A_1$  or like  $A_2$  if both  $c_1$  and  $c_2$  are entailed by the current store, it behaves like  $A_i$  if  $c_i$  only is entailed, it suspends if both  $c_1$  and  $c_2$  are consistent with but not entailed by the current store, and it behaves like “ $\text{ask}(c_1) \rightarrow A_1$ ” whenever “ $\text{ask}(c_2) \rightarrow A_2$ ” fails (and vice versa);
- agent “ $\text{tell}(c) \rightarrow A$ ” adds constraint  $c$  to the current store and then, if the resulting store is consistent, behaves like  $A$ , otherwise it fails.
- agent  $A_1 \parallel A_2$  behaves like  $A_1$  and  $A_2$  executing in parallel;
- agent  $\exists_x A$  behaves like agent  $A$ , except that the variables in  $x$  are local to  $A$ ;
- $p(x)$  is a call of procedure  $p$ .

A formal treatment of the cc semantics can be found in [Saraswat 1993; de Boer and Palamidessi 1991]. Also, a denotational semantics of deterministic cc programs, based on closure operators, can be found in [Saraswat 1993]. A more complete survey on several concurrent paradigms is given also in [de Boer and Palamidessi 1994].

## 2.2 Soft Constraints

Several formalization of the concept of *soft constraints* are currently available. In the following, we refer to the one based on c-semirings [Bistarelli et al. 1997; Bistarelli 2001], which can be shown to generalize and express many of the others.

A soft constraint may be seen as a constraint where each instantiations of its variables has an associated value from a partially ordered set which can be interpreted as a set of preference values. Combining constraints will then have to take into account such additional values, and thus the formalism has also to provide suitable operations for combination ( $\times$ ) and comparison ( $+$ ) of tuples of values and

constraints. This is why this formalization is based on the concept of c-semiring, which is just a set plus two operations.

2.2.1 *C-semirings*. A semiring is a tuple  $\langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$  such that:

- (1)  $A$  is a set and  $\mathbf{0}, \mathbf{1} \in A$ ;
- (2)  $+$  is commutative, associative and  $\mathbf{0}$  is its unit element;
- (3)  $\times$  is associative, distributes over  $+$ ,  $\mathbf{1}$  is its unit element and  $\mathbf{0}$  is its absorbing element.

A *c-semiring* is a semiring  $\langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$  such that  $+$  is idempotent,  $\mathbf{1}$  is its absorbing element and  $\times$  is commutative. Let us consider the relation  $\leq_S$  over  $A$  such that  $a \leq_S b$  iff  $a + b = b$ . Then it is possible to prove that (see [Bistarelli et al. 1997]):

- (1)  $\leq_S$  is a partial order;
- (2)  $+$  and  $\times$  are monotone on  $\leq_S$ ;
- (3)  $\mathbf{0}$  is its minimum and  $\mathbf{1}$  its maximum;
- (4)  $\langle A, \leq_S \rangle$  is a complete lattice and, for all  $a, b \in A$ ,  $a + b = \text{lub}(a, b)$ .

Moreover, if  $\times$  is idempotent, then:  $+$  distribute over  $\times$ ;  $\langle A, \leq_S \rangle$  is a complete distributive lattice and  $\times$  its glb. Informally, the relation  $\leq_S$  gives us a way to compare semiring values and constraints. In fact, when we have  $a \leq_S b$ , we will say that  $b$  is *better than*  $a$ . In the following, when the semiring will be clear from the context,  $a \leq_S b$  will be often indicated by  $a \leq b$ .

2.2.2 *Soft Constraints and Problems*. Given a semiring  $S = \langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$ , a finite set  $D$  (the domain of the variables) and an ordered set of variables  $V$ , a *constraint* is a pair  $\langle \text{def}, \text{con} \rangle$  where  $\text{con} \subseteq V$  and  $\text{def} : D^{|\text{con}|} \rightarrow A$ . Therefore, a constraint specifies a set of variables (the ones in  $\text{con}$ ), and assigns to each tuple of values of these variables an element of the semiring. Consider two constraints  $c_1 = \langle \text{def}_1, \text{con} \rangle$  and  $c_2 = \langle \text{def}_2, \text{con} \rangle$ , with  $|\text{con}| = k$ . Then  $c_1 \sqsubseteq_S c_2$  if for all  $k$ -tuples  $t$ ,  $\text{def}_1(t) \leq_S \text{def}_2(t)$ . The relation  $\sqsubseteq_S$  is a partial order.

A *soft constraint problem* is a pair  $\langle C, \text{con} \rangle$  where  $\text{con} \subseteq V$  and  $C$  is a set of constraints:  $\text{con}$  is the set of variables of interest for the constraint set  $C$ , which however may concern also variables not in  $\text{con}$ . Note that a classical CSP is a SCSP where the chosen c-semiring is:  $S_{\text{CSP}} = \langle \{\text{false}, \text{true}\}, \vee, \wedge, \text{false}, \text{true} \rangle$ . Fuzzy CSPs [Schiex 1992] can instead be modeled in the SCSP framework by choosing the c-semiring  $S_{\text{FCSP}} = \langle [0, 1], \max, \min, 0, 1 \rangle$ . Many other “soft” CSPs (Probabilistic, weighted, ...) can be modeled by using a suitable semiring structure (for example,  $S_{\text{prob}} = \langle [0, 1], \max, \times, 0, 1 \rangle$ ,  $S_{\text{weight}} = \langle \mathcal{R}, \min, +, 0, +\infty \rangle, \dots$ ).

Figure 1 shows the graph representation of a fuzzy CSP. Variables and constraints are represented respectively by nodes and by undirected arcs (unary for  $c_1$  and  $c_3$  and binary for  $c_2$ ), and semiring values are written to the right of the corresponding tuples. The variables of interest (that is the set  $\text{con}$ ) are represented with a double circle. Here we assume that the domain  $D$  of the variables contains only elements  $a$  and  $b$ .

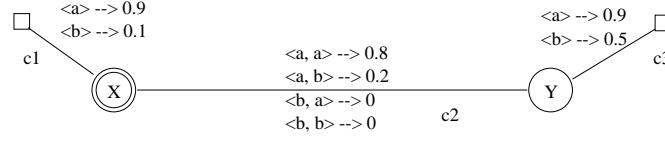


Fig. 1. A fuzzy CSP

**2.2.3 Combining and projecting soft constraints.** Given two constraints  $c_1 = \langle def_1, con_1 \rangle$  and  $c_2 = \langle def_2, con_2 \rangle$ , their *combination*  $c_1 \otimes c_2$  is the constraint  $\langle def, con \rangle$  defined by  $con = con_1 \cup con_2$  and  $def(t) = def_1(t \downarrow_{con_1}^{con}) \times def_2(t \downarrow_{con_2}^{con})$ , where  $t \downarrow_Y^X$  denotes the tuple of values over the variables in  $Y$ , obtained by projecting tuple  $t$  from  $X$  to  $Y$ . In words, combining two constraints means building a new constraint involving all the variables of the original ones, and which associates to each tuple of domain values for such variables a semiring element which is obtained by multiplying the elements associated by the original constraints to the appropriate subtuples.

Given a constraint  $c = \langle def, con \rangle$  and a subset  $I$  of  $V$ , the *projection* of  $c$  over  $I$ , written  $c \downarrow_I$  is the constraint  $\langle def', con' \rangle$  where  $con' = con \cap I$  and  $def'(t') = \sum_{t/t \downarrow_{I \cap con}^{con} = t'} def(t)$ . Informally, projecting means eliminating some variables. This is done by associating to each tuple over the remaining variables a semiring element which is the sum of the elements associated by the original constraint to all the extensions of this tuple over the eliminated variables. In short, combination is performed via the multiplicative operation of the semiring, and projection via the additive one.

**2.2.4 Solutions.** The *solution* of an SCSP problem  $P = \langle C, con \rangle$  is the constraint  $Sol(P) = (\bigotimes C) \downarrow_{con}$ . That is, we combine all constraints, and then project over the variables in  $con$ . In this way we get the constraint over  $con$  which is “induced” by the entire SCSP.

For example, the solution of the fuzzy CSP of Figure 1 associates a semiring element to every domain value of variable  $x$ . Such an element is obtained by first combining all the constraints together. For instance, for the tuple  $\langle a, a \rangle$  (that is,  $x = y = a$ ), we have to compute the minimum between 0.9 (which is the value assigned to  $x = a$  in constraint  $c_1$ ), 0.8 (which is the value assigned to  $\langle x = a, y = a \rangle$  in  $c_2$ ) and 0.9 (which is the value for  $y = a$  in  $c_3$ ). Hence, the resulting value for this tuple is 0.3. We can do the same work for tuple  $\langle a, b \rangle \rightarrow 0.2$ ,  $\langle b, a \rangle \rightarrow 0$  and  $\langle b, b \rangle \rightarrow 0$ . The obtained tuples are then projected over variable  $x$ , obtaining the solution  $\langle a \rangle \rightarrow 0.8$  and  $\langle b \rangle \rightarrow 0$ .

Sometimes it may be useful to find only a semiring value representing the least upper bound among the values yielded by the solutions. This is called the *best level of consistency* of an SCSP problem  $P$  and it is defined by  $blevel(P) = Sol(P) \downarrow_\emptyset$  (for instance, the fuzzy CSP of Figure 1 has best level of consistency 0.8). We also say that:  $P$  is  $\alpha$ -consistent if  $blevel(P) = \alpha$ ;  $P$  is consistent iff there exists  $\alpha > 0$  such that  $P$  is  $\alpha$ -consistent;  $P$  is inconsistent if it is not consistent.

### 3. CONCURRENT CONSTRAINT PROGRAMMING OVER SOFT CONSTRAINTS

Given a semiring  $S = \langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$  and an ordered set of variables  $V$  over a finite domain  $D$ , we will now show how soft constraints over  $S$  with a suitable pair of operators form a semiring, and then, we highlight the properties needed to map soft constraints over constraint systems “*a la Saraswat*” (as recalled in Section 2.1).

We start by giving the definition of the carrier set of the semiring.

*Definition 3.1 (functional constraints).* We define  $\mathcal{C} = (V \rightarrow D) \rightarrow A$  as the set of all possible constraints that can be built starting from  $S = \langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$ ,  $D$  and  $V$ .

A generic function describing the assignment of domain elements to variables will be denoted in the following by  $\eta : V \rightarrow D$ . Thus a constraint is a function which, given an assignment  $\eta$  of the variables, returns a value of the semiring.

Note that in this *functional* formulation, each constraint is a function and not a pair representing the variable involved and its definition. Such a function involves all the variables in  $V$ , but it depends on the assignment of only a finite subset of them. We call this subset the *support* of the constraint. For computational reasons we require each support to be finite.

*Definition 3.2 (constraint support).* Consider a constraint  $c \in \mathcal{C}$ . We define his support as  $\text{supp}(c) = \{v \in V \mid \exists \eta, d_1, d_2. c\eta[v := d_1] \neq c\eta[v := d_2]\}$ , where

$$\eta[v := d]v' = \begin{cases} d & \text{if } v = v', \\ \eta v' & \text{otherwise.} \end{cases}$$

Note that  $c\eta[v := d_1]$  means  $c\eta'$  where  $\eta'$  is  $\eta$  modified with the association  $v := d_1$  (that is the operator  $[\ ]$  has precedence over application).

*Definition 3.3 (functional mapping).* Given any soft constraint  $\langle \text{def}, \{v_1, \dots, v_n\} \rangle \in C$ , we can define its corresponding function  $c \in \mathcal{C}$  s.t.  $c\eta[v_1 := d_1] \dots [v_n := d_n] = \text{def}(d_1, \dots, d_n)$ . Clearly  $\text{supp}(c) \subseteq \{v_1, \dots, v_n\}$ .

*Definition 3.4 (Combination and Sum).* Given the set  $\mathcal{C}$ , we can define the combination and sum functions  $\otimes, \oplus : \mathcal{C} \times \mathcal{C} \rightarrow \mathcal{C}$  as follows:

$$(c_1 \otimes c_2)\eta = c_1\eta \times_S c_2\eta \quad \text{and} \quad (c_1 \oplus c_2)\eta = c_1\eta +_S c_2\eta.$$

Notice that function  $\otimes$  has the same meaning of the already defined  $\otimes$  operator (see Section 2.2) while function  $\oplus$  models a sort of disjunction.

By using the  $\oplus_S$  operator we can easily extend the partial order  $\leq_S$  over  $\mathcal{C}$  by defining  $c_1 \sqsubseteq_S c_2 \iff c_1 \oplus_S c_2 = c_2$ . In the following, when the semiring will be clear from the context, we will use  $\sqsubseteq$ .

We can also define a unary operator that will be useful to represent the unit elements of the two operations  $\oplus$  and  $\otimes$ . To do that, we need the definition of constant functions over a given set of variables.

*Definition 3.5 (constant function).* We define function  $\bar{a}$  as the function that returns the semiring value  $a$  for all assignments  $\eta$ , that is,  $\bar{a}\eta = a$ . We will usually write  $\bar{a}$  simply as  $a$ .



An example of constants that will be useful later are  $\bar{\mathbf{0}}$  and  $\bar{\mathbf{1}}$  that represent respectively the constraint associating  $\mathbf{0}$  and  $\mathbf{1}$  to all the assignment of domain values.

It is easy to verify that each constant has an empty support. More generally we can prove the following:

**PROPOSITION 3.6.** *The support of a constraint  $c \Downarrow_I$  is always a subset of  $I$  (that is  $\text{supp}(c \Downarrow_I) \subseteq I$ ).*

**PROOF.** By definition of  $\Downarrow_I$ , for any variable  $x \notin I$  we have  $c \Downarrow_I \eta[x = a] = c \Downarrow_I \eta[x = b]$  for any  $a$  and  $b$ . So, by definition of support  $x \notin \text{supp}(c \Downarrow_I)$ .  $\square$

**THEOREM 3.7 (HIGHER ORDER SEMIRING).** *The structure  $S_C = \langle \mathcal{C}, \oplus, \otimes, \mathbf{0}, \mathbf{1} \rangle$  where*

- $\mathcal{C} : (V \rightarrow D) \rightarrow A$  *is the set of all the possible constraints that can be built starting from  $S$ ,  $D$  and  $V$  as defined in Definition 3.1,*
  - $\otimes$  *and  $\oplus$  are the functions defined in Definition 3.4, and*
  - $\mathbf{0}$  *and  $\mathbf{1}$  are constant functions defined following Definition 3.5,*
- is a c-semiring.*

**PROOF.** To prove the theorem it is enough to check all the properties with the fact that the same properties hold for semiring  $S$ . We give here only a hint, by showing the commutativity of the  $\otimes$  operator:

$$\begin{aligned} c_1 \otimes c_2 \eta &= (\text{by definition of } \otimes) \\ c_1 \eta \times c_2 \eta &= (\text{by commutativity of } \times) \\ c_2 \eta \times c_1 \eta &= (\text{by definition of } \otimes) \\ c_2 \otimes c_1 \eta. \end{aligned}$$

All the other properties can be proved similarly.  $\square$

The next step is to look for a notion of token and of entailment relation. We define as tokens the functional constraints in  $\mathcal{C}$  and we introduce a relation  $\vdash$  that is an entailment relation when the multiplicative operator of the semiring is idempotent.

**Definition 3.8 ( $\vdash$  relation).** Consider the high order semiring carrier set  $\mathcal{C}$  and the partial order  $\sqsubseteq$ . We define the relation  $\vdash \subseteq \wp(\mathcal{C}) \times \mathcal{C}$  s.t. for each  $C \in \wp(\mathcal{C})$  and  $c \in \mathcal{C}$ , we have  $C \vdash c \iff \bigotimes C \sqsubseteq c$ .

The next theorem shows that, when the multiplicative operator of the semiring is idempotent, the  $\vdash$  relation satisfies all the properties needed by an entailment.

**THEOREM 3.9 ( $\vdash$  WITH IDEMPOTENT  $\times$  IS AN ENTAILMENT RELATION).** *Consider the higher order semiring carrier set  $\mathcal{C}$  and the partial order  $\sqsubseteq$ . Consider also the relation  $\vdash$  of Definition 3.8. Then, if the multiplicative operation of the semiring is idempotent,  $\vdash$  is an entailment relation.*

**PROOF.** Is enough to check that for any  $c \in \mathcal{C}$ , and for any  $C_1, C_2$  and  $C_3$  subsets of  $\mathcal{C}$  we have

- (1)  $C \vdash c$  when  $c \in C$ : We need to show that  $\bigotimes C \sqsubseteq c$  when  $c \in C$ . This follows from the extensivity of  $\times$ .

- (2) **if**  $C_1 \vdash C_2$  **and**  $C_2 \vdash C_3$  **then**  $C_1 \vdash C_3$ : To prove this we use the extended version of the relation  $\vdash$  able to deal with subsets of  $\mathcal{C} : \wp(\mathcal{C}) \times \wp(\mathcal{C})$  s.t.  $C_1 \vdash C_2 \iff C_1 \vdash \otimes C_2$ . Note that when  $\times$  is idempotent we have that,  $\forall c_2 \in C_2, C_1 \vdash c_2 \iff C_1 \vdash \otimes C_2$ . In this case to prove the item we have to prove that if  $\otimes C_1 \sqsubseteq \otimes C_2$  and  $\otimes C_2 \sqsubseteq \otimes C_3$ , then  $\otimes C_1 \sqsubseteq \otimes C_3$ . This comes from the transitivity of  $\sqsubseteq$ .

□

Note that in this setting the notion of token (constraint) and of set of tokens (set of constraints) closed under entailment is used indifferently. In fact, given a set of constraint functions  $C_1$ , its closure w.r.t. entailment is a set  $\bar{C}_1$  that contains all the constraints greater than  $\otimes C_1$ . This set is univocally representable by the constraint function  $\otimes C_1$ .

The definition of the entailment operator  $\vdash$  on top of the higher order semiring  $S_C = \langle \mathcal{C}, \oplus, \otimes, \mathbf{0}, \mathbf{1} \rangle$  and of the  $\sqsubseteq$  relation leads to the notion of *soft constraint system*. It is also important to notice that in [Saraswat 1993] it is claimed that a constraint system is a *complete algebraic* lattice. Here we do not ask for this, since the algebraic nature of the structure  $\mathcal{C}$  strictly depends on the properties of the semiring.

### 3.1 Non-idempotent $\times$

If the constraint system is defined on top of a non-idempotent multiplicative operator, we cannot obtain a  $\vdash$  relation satisfying all the properties of an entailment. Nevertheless, we can give a *denotational* semantics to the constraint store, as described in Section 4, using the operations of the higher order semiring.

To treat the hiding operator of the language, a general notion of existential quantifier has to be introduced by using notions similar to those used in cylindric algebras. Note however that cylindric algebras are first of all boolean algebras. This could be possible in our framework only when the  $\times$  operator is idempotent.

*Definition 3.10 (hiding).* Consider a set of variables  $V$  with domain  $D$  and the corresponding soft constraint system  $\mathcal{C}$ . We define for each  $x \in V$  the hiding function  $(\exists_x c)\eta = \sum_{d_i \in D} c\eta[x := d_i]$ .

Notice that  $x$  does not belong to the support of  $\exists_x c$ .

By using the hiding function we can represent the  $\Downarrow$  operator defined in Section 2.2.

**PROPOSITION 3.11.** *Consider a semiring  $S = \langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$ , a domain of the variables  $D$ , an ordered set of variables  $V$ , the corresponding structure  $\mathcal{C}$  and the class of hiding functions  $\exists_x : \mathcal{C} \rightarrow \mathcal{C}$  as defined in Definition 3.10. Then, for any constraint  $c$  and any variable  $x \subseteq V$ ,  $c \Downarrow_{V-x} = \exists_x c$ .*

**PROOF.** Is enough to apply the definition of  $\Downarrow_{V-x}$  and  $\exists_x$  and check that both are equal to  $\sum_{d_i \in D} c\eta[x := d_i]$ . □

We now show how the hiding function so defined satisfies the properties of cylindric algebras.

**THEOREM 3.12.** *Consider a semiring  $S = \langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$ , a domain of the variables  $D$ , an ordered set of variables  $V$ , the corresponding structure  $\mathcal{C}$  and the class of hiding functions  $\exists_x : \mathcal{C} \rightarrow \mathcal{C}$  as defined in Definition 3.10. Then  $\mathcal{C}$  is a cylindric algebra satisfying:*

- (1)  $c \vdash \exists_x c$
- (2)  $c_1 \vdash c_2$  implies  $\exists_x c_1 \vdash \exists_x c_2$
- (3)  $\exists_x (c_1 \otimes \exists_x c_2) \approx \exists_x c_1 \otimes \exists_x c_2$ ,
- (4)  $\exists_x \exists_y c \approx \exists_y \exists_x c$

**PROOF.** Let us consider all the items:

- (1) It follows from the intensivity of  $+$ ;
- (2) It follows from the monotonicity of  $+$ ;
- (3) It follows from theorems about distributivity and idempotence, proven in [Bistarelli et al. 1997];
- (4) It follows from commutativity and associativity of  $+$ .

□

To model parameter passing we need also to define what diagonal elements are.

**Definition 3.13 (diagonal elements).** Consider an ordered set of variables  $V$  and the corresponding soft constraint system  $\mathcal{C}$ . Let us define for each  $x, y \in V$  a constraint  $d_{xy} \in \mathcal{C}$  s.t.,  $d_{xy}\eta[x := a, y := b] = \mathbf{1}$  if  $a = b$  and  $d_{xy}\eta[x := a, y := b] = \mathbf{0}$  if  $a \neq b$ . Notice that  $\text{supp}(d_{xy}) = \{x, y\}$ .

We can prove that the constraints just defined are diagonal elements.

**THEOREM 3.14.** *Consider a semiring  $S = \langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$ , a domain of the variables  $D$ , an ordered set of variables  $V$ , and the corresponding structure  $\mathcal{C}$ . The constraints  $d_{xy}$  defined in Definition 3.13 represent diagonal elements, that is*

- (1)  $d_{xx} = \mathbf{1}$ ,
- (2) if  $z \neq x, y$  then  $d_{xy} = \exists_z (d_{xz} \otimes d_{zy})$ ,
- (3) if  $x \neq y$  then  $d_{xy} \otimes \exists_x (c \otimes d_{xy}) \vdash c$ .

**PROOF.** (1) It follows from the definition of the  $\mathbf{1}$  constant and of the diagonal constraint;

- (2) The constraint  $d_{xz} \otimes d_{zy}$  is equal to  $\mathbf{1}$  when  $x = y = z$ , and is equal to  $\mathbf{0}$  in all the other cases. If we project this constraint over  $z$ , we obtain the constraint  $\exists_z (d_{xz} \otimes d_{zy})$  that is equal to  $\mathbf{1}$  only when  $x = y$ ;
- (3) The constraint  $(c \otimes d_{xy})\eta$  has value  $\mathbf{0}$  whenever  $\eta(x) \neq \eta(y)$  and  $c\eta$  elsewhere. Now,  $(\exists_x (c \otimes d_{xy}))\eta$  is by definition equal to  $c\eta[x := y]$ . Thus  $(d_{xy} \otimes \exists_x (c \otimes d_{xy}))\eta$  is equal to  $c\eta$  when  $\eta(x) = \eta(y)$  and  $\mathbf{0}$  elsewhere. By the last assumption, we have the relation of entailment with  $c$ .

□

Table II. scc syntax

---

$P ::= F.A$
$F ::= p(X) :: A \mid F.F$
$A ::= stop \mid tell(c) \rightarrow_{\phi} A \mid tell(c) \rightarrow^a A \mid E \mid A \parallel A \mid \exists X.A \mid p(X)$
$E ::= ask(c) \rightarrow_{\phi} A \mid ask(c) \rightarrow^a A \mid E + E$

---

### 3.2 Using cc on top of a soft constraint system

The only problem in using a soft constraint system in a cc language is the interpretation of the *consistency* notion necessary to deal with the ask and tell operations.

Usually SCSPs with best level of consistency equal to  $\mathbf{0}$  are interpreted as inconsistent, and those with level greater than  $\mathbf{0}$  as consistent, but we can be more general. In fact, we can define a suitable function  $\alpha$  that, given the best level of the actual store, will map such a level over the classical notion of consistency/inconsistency. More precisely, given a semiring  $S = \langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$ , we can define a function  $\alpha : A \rightarrow \{false, true\}$ . Function  $\alpha$  has to be at least monotone, but functions with a richer set of properties could be used. It is worth to notice that in a different environment some of the authors use a similar function to map elements from a semiring to another, by using abstract interpretation techniques [Bistarelli et al. 2000a; 2000b]

Whenever we need to check the consistency of the store, we will first compute the best level and then we will map such a value by using function  $\alpha$  over *true* or *false*.

It is important to notice that changing the  $\alpha$  function (that is, by mapping in a different way the set of values  $A$  over the boolean elements *true* and *false*), the same cc agent yields different results: by using a high cut level, the cc agent will either finish with a failure or succeed with a high final best level of consistency of the store. On the other hand, by using a low level, more programs will end in a success state.

## 4. SOFT CONCURRENT CONSTRAINT PROGRAMMING

The next step in our work is now to extend the syntax of the language in order to directly handle the cut level. This means that the syntax and semantics of the tell and ask agents have to be enriched with a threshold to specify when tell/ask agents have to fail, succeed or suspend.

Given a soft constraint system  $\langle S, D, V \rangle$  and the corresponding structure  $\mathcal{C}$ , and any constraint  $\phi \in \mathcal{C}$ , the syntax of agents in soft concurrent constraint programming is given in Table II. The main difference w.r.t. the original cc syntax is the presence of a semiring element  $a$  and of a constraint  $\phi$  to be checked whenever an *ask* or *tell* operation is performed. More precisely, the level  $a$  (resp.,  $\phi$ ) will be used as a cut level to prune computations that are not good enough.

We present here a structured operational semantics for scc programs, in the SOS style, which consists of defining the semantic of the programming language by specifying a set of *configurations*  $\Gamma$ , which define the states during execution, a relation  $\rightarrow \subseteq \Gamma \times \Gamma$  which describes the *transition* relation between the configurations, and a set  $T$  of *terminal* configurations. To give an operational semantics to our language,

Table III. Transition rules for scc

$\langle stop, \sigma \rangle \longrightarrow \langle success, \sigma \rangle$	(Stop)
$\frac{(\sigma \otimes c) \Downarrow_{\emptyset} \not\prec a}{\langle tell(c) \rightarrow^a A, \sigma \rangle \longrightarrow \langle A, \sigma \otimes c \rangle}$	(Valued-tell)
$\frac{\sigma \otimes c \not\sqsubseteq \phi}{\langle tell(c) \rightarrow_{\phi} A, \sigma \rangle \longrightarrow \langle A, \sigma \otimes c \rangle}$	(Tell)
$\frac{\sigma \vdash c, \sigma \Downarrow_{\emptyset} \not\prec a}{\langle ask(c) \rightarrow^a A, \sigma \rangle \longrightarrow \langle A, \sigma \rangle}$	(Valued-ask)
$\frac{\sigma \vdash c, \sigma \not\sqsubseteq \phi}{\langle ask(c) \rightarrow_{\phi} A, \sigma \rangle \longrightarrow \langle A, \sigma \rangle}$	(Ask)
$\frac{\langle A_1, \sigma \rangle \longrightarrow \langle A'_1, \sigma' \rangle \quad \langle A_1, \sigma \rangle \longrightarrow \langle success, \sigma' \rangle}{\langle A_1 \parallel A_2, \sigma \rangle \longrightarrow \langle A'_1 \parallel A_2, \sigma' \rangle \quad \langle A_1 \parallel A_2, \sigma \rangle \longrightarrow \langle A_2, \sigma' \rangle}$	(Parallelism)
$\frac{\langle A_2 \parallel A_1, \sigma \rangle \longrightarrow \langle A_2 \parallel A'_1, \sigma' \rangle \quad \langle A_2 \parallel A_1, \sigma \rangle \longrightarrow \langle A_2, \sigma' \rangle}{\langle A_2 \parallel A_1, \sigma \rangle \longrightarrow \langle A_2, \sigma' \rangle}$	
$\frac{\langle E_1, \sigma \rangle \longrightarrow \langle A_1, \sigma' \rangle}{\langle E_1 + E_2, \sigma \rangle \longrightarrow \langle A_1, \sigma' \rangle \quad \langle E_2 + E_1, \sigma \rangle \longrightarrow \langle A_1, \sigma' \rangle}$	(Nondeterminism)
$\frac{\langle A[y/x], \sigma \rangle \longrightarrow \langle A', \sigma' \rangle}{\langle \exists_x A, \sigma \rangle \longrightarrow \langle A', \sigma' \rangle}$ with $y$ fresh	(Hidden variables)
$\langle p(y), \sigma \rangle \longrightarrow \langle A[y/x], \sigma \rangle$ when $p(x) :: A$	(Procedure call)

we need to describe an appropriate transition system.

*Definition 4.1 (transition system).* A transition system is a triple  $\langle \Gamma, T, \rightarrow \rangle$  where  $\Gamma$  is a set of possible configurations,  $T \subseteq \Gamma$  is the set of *terminal* configurations and  $\rightarrow \subseteq \Gamma \times \Gamma$  is a binary relation between configurations.

The set of configurations represent the evolutions of the agents and the modifications in the constraint store. We define the transition system of soft cc as follows:

*Definition 4.2 (configurations).* The set of configurations for a soft cc system is the set  $\Gamma = \{ \langle A, \sigma \rangle \} \cup \{ \langle success, \sigma \rangle \}$ , where  $\sigma \in \mathcal{C}$ . The set of terminal configurations is the set  $T = \{ \langle success, \sigma \rangle \}$  and the transition rule for the scc language are defined in Table III.

Here is a brief description of the transition rules:

*Stop.* The stop agent succeeds in one step by transforming itself into terminal configuration *success*.

*Valued-tell.* The valued-tell rule checks for the  $\alpha$ -consistency of the SCSP defined by the store  $\sigma \otimes c$ . The rule can be applied only if the store  $\sigma \otimes c$  is  $b$ -consistent with  $b \not\prec a$ . In this case the agent evolves to the new agent  $A$  over the store  $\sigma \otimes c$ . Note that different choices of the *cut level*  $a$  could possibly lead to different computations.

*Tell.* The tell action is a finer check of the store. In this case, a pointwise comparison between the store  $\sigma \otimes c$  and the constraint  $\phi$  is performed. The idea is to

perform an overall check of the store and to continue the computation only if there is the possibility to compute a solution not worse than  $\phi$ .

*Valued-ask.* The semantics of the valued-ask is extended in a way similar to what we have done for the valued-tell action. This means that, to apply the rule, we need to check if the store  $\sigma$  entails the constraint  $c$  and also if the store is “consistent enough” w.r.t. the threshold  $a$  set by the programmer.

*Ask.* Similar to the *tell* rule, here a finer (pointwise) threshold  $\phi$  is compared to the store  $\sigma$ .

*Nondeterminism and parallelism.* The composition operators  $+$  and  $\parallel$  are not modified w.r.t. the classical ones: a parallel agent will succeed if all the agents succeeds; a nondeterministic rule chooses any agent whose guard succeeds.

*Hidden variables.* The semantics of the existential quantifier is similar to that described in [Saraswat 1993] by using the notion of *freshness* of the new variable added to the store.

*Procedure calls.* The semantics of the procedure call is not modified w.r.t. the classical one. The only difference is the different use of the diagonal constraint to represent parameter passing.

#### 4.1 Eventual Tell/Ask

We recall that both ask and tell operations in cc could be either atomic (that is, if the corresponding check is not satisfied, the agent does not evolve) or eventual (that is, the agent evolves regardless of the result of the check). It is interesting to notice that the transition rules defined in Table III could be used to provide both interpretations of the ask and tell operations. In fact, while the generic tell/ask rule represents an atomic behaviour, by setting  $\phi = \mathbf{0}$  or  $a = \mathbf{0}$  we obtain their *eventual* version:

$$\begin{array}{l} \langle \text{tell}(c) \rightarrow A, \sigma \rangle \longrightarrow \langle A, \sigma \otimes c \rangle \quad \text{(Eventual tell)} \\ \hline \sigma \vdash c \quad \text{(Eventual ask)} \\ \langle \text{ask}(c) \rightarrow A, \sigma \rangle \longrightarrow \langle A, \sigma \rangle \end{array}$$

Notice that, by using an eventual interpretation, the transition rules of the scc become the same as those of cc (with an eventual interpretation too). This happens since, in the eventual version, the tell/ask agent never checks for consistency and so the soft notion of  $\alpha$ -consistency does not play any role.

### 5. A SIMPLE EXAMPLE

In this section we will show the behaviour of some of the rules of our transition system. We consider in this example a soft constraint system over the fuzzy semiring. Consider the fuzzy constraints

$$\begin{array}{ll} c : \{x, y\} \rightarrow \mathbb{R}^2 \rightarrow [0, 1] & \text{s.t. } c(x, y) = \frac{1}{1 + |x - y|} \quad \text{and} \\ c' : \{x\} \rightarrow \mathbb{R} \rightarrow [0, 1] & \text{s.t. } c'(x) = \begin{cases} 1 & \text{if } x \leq 10, \\ 0 & \text{otherwise.} \end{cases} \end{array}$$

Notice that the domain of both variables  $x$  and  $y$  is in this example any integer (or real) number. As any fuzzy CSP, the definition of the constraints is instead in the interval  $[0, 1]$ .

Let's now evaluate the agent

$$\langle tell(c) \rightarrow^{0.4} ask(c') \rightarrow^{0.8} stop, 1 \rangle$$

in the empty starting store 1.

By applying the *Valued-tell* rule we need to check  $(1 \otimes c) \Downarrow_{\emptyset} \not\leq 0.4$ . Since  $1 \otimes c = c$  and  $c \Downarrow_{\emptyset} = 1$ , the agent can perform the step, and it reaches the state

$$\langle ask(c') \rightarrow^{0.8} stop, c \rangle.$$

Now we need to check (by following the rule of *Valued-ask*) if  $c \vdash c'$  and  $c \Downarrow_{\emptyset} \not\leq 0.8$ . While the second relation easily holds, the first one does not hold (in fact, for  $x = 11$  and  $y = 10$  we have  $c'(x) = 0$  and  $c(x, y) = 0.5$ ).

If instead we consider the constraint  $c''(x, y) = \frac{1}{1+2 \times |x-y|}$  in place of  $c'$ , then we have

$$\langle ask(c'') \rightarrow^{0.8} stop, c \rangle.$$

Here the condition  $c \vdash c''$  easily holds and the agent  $ask(c'') \rightarrow^{0.8} stop$  can perform its last step, reaching the *stop* and *success* states:

$$\langle stop, c \otimes c'' \rangle \rightarrow \langle success, c \otimes c'' \rangle.$$

## 6. OBSERVABLES AND CUTS

Sometimes one could desire to see an agent, and a corresponding program, execute with a cut level which is different from the one originally given. We will therefore define  $cut_{\psi}(A)$  the agent  $A$  where all the occurrences of any cut level, say  $\phi$ , in any subagent of  $A$  or in any clause of the program, are replaced by  $\psi$  if  $\phi \sqsubseteq \psi$ . This means that the cut level of each subagent and clause becomes at least  $\psi$ , or is left to the original level.

In this paper, for simplicity and generality reasons, this cut level change applies only to those programs with cut levels which are constraints ( $\phi$ ), and not single semiring levels ( $a$ ).

*Definition 6.1 (cut function).* Consider an scc agent  $A$ ; we define the function  $cut_{\psi} : A \rightarrow A$  that transforms ask and tell subagents as follows:

$$cut_{\psi}(ask/tell(c) \rightarrow_{\phi}) = \begin{cases} ask/tell(c) \rightarrow_{\psi} & \text{if } \phi \sqsubseteq \psi, \\ ask/tell(c) \rightarrow_{\phi} & \text{otherwise.} \end{cases}$$

By definition of  $cut_{\psi}$ , it is easy to see that  $cut_0(A) = A$ .

We can then prove the following Lemma (that will be useful later):

**LEMMA 6.2 (TELL AND ASK CUT).** *Consider the Tell and Ask rules of Table III, and the constraints  $\sigma$  and  $c$  as defined in such rules. Then:*

—*If the Tell rule can be applied to agent  $A$ , then the rule can be applied also to  $cut_{\psi}(A)$  when  $\psi \sqsubseteq \sigma \otimes c$ .*

—If the Ask rule can be applied to agent  $A$ , then the rule can be applied also to  $cut_\psi(A)$  when  $\psi \sqsubseteq \sigma$ .

PROOF. We will prove only the first item; the second can be easily proved by using the same ideas. By the definition of the tell transition rules of Table III, if we can apply the rule it means that  $A ::= tell(c) \rightarrow_\phi A'$  and if  $\sigma$  is the store we have  $\sigma \otimes c \not\sqsubseteq \phi$ . Now, by definition of  $cut_\psi$ , we can have

— $cut_\psi(A) ::= tell(c) \rightarrow_\psi cut_\psi(A')$  when  $\phi \sqsubseteq \psi$ .

— $cut_\psi(A) ::= tell(c) \rightarrow_\phi cut_\psi(A')$  when  $\phi \not\sqsubseteq \psi$ ,

In the first case, the statement holds by initial hypothesis over  $A$ . In the second case, since by hypothesis we have  $\sigma \otimes c \not\sqsubseteq \psi$ , again the statement holds by the definition of the tell transition rules of Table III.  $\square$

It is now interesting to notice that the thresholds appearing in the program are related to the final computed stores:

THEOREM 6.3 (THRESHOLDS). *Consider an scc computation*

$$\langle A, \mathbf{1} \rangle \rightarrow \langle A_1, \sigma_1 \rangle \rightarrow \dots \langle A_n, \sigma_n \rangle \rightarrow \langle success, \sigma \rangle$$

for a program  $P$ . Then, also

$$\langle cut_\sigma(A), \mathbf{1} \rangle \rightarrow \langle cut_\sigma(A_1), \sigma_1 \rangle \rightarrow \dots \langle cut_\sigma(A_n), \sigma_n \rangle \rightarrow \langle success, \sigma \rangle$$

is an scc computation for program  $P$ .

PROOF. First of all, notice that during the computation an agent can only add constraints to the store. So, since  $\times$  is extensive, the store can only monotonically decrease starting from the initial store  $\mathbf{1}$  and ending in the final store  $\sigma$ . So we have

$$\mathbf{1} \sqsupseteq \sigma_1 \dots \sqsupseteq \sigma_n \sqsupseteq \sigma.$$

Now, the statement follows by applying at each step the results of Lemma 6.2. In fact, at each step the hypothesis of the lemma hold:

—the cut  $\sigma$  is always lower than the current store ( $\sigma \sqsubseteq \sigma_i \otimes c$ );

—the ask and tell operations can be applied (moving from agent  $A_i$  to agent  $A_i + 1$ ).

$\square$

### 6.1 Capturing Success Computations.

Given the transition system as defined in the previous section, we now define what we want to observe of the program behaviour as described by the transitions. To do this, we define for each agent  $A$  the set of constraints

$$\mathcal{S}_A = \{\sigma \downarrow_{var(A)} \mid \langle A, \mathbf{1} \rangle \rightarrow^* \langle success, \sigma \rangle\}$$

that collects the results of the successful computations that the agent can perform. Notice that the computed store  $\sigma$  is projected over the variables of the agent  $A$  to discard any *fresh* variable introduced in the store by the  $\exists$  operator.

The observable  $\mathcal{S}_A$  could be refined by considering, instead of the set of successful computations starting from  $\langle A, \mathbf{1} \rangle$ , only a subset of them. For example,



one could be interested in considering only the *best* computations: in this case, all the computations leading to a store worse than one already collected are disregarded. With a pessimistic view, the representative subset could instead collect all the worst computations (that is, all the computations better than others are disregarded). Finally, also a set containing both the best and the worst computations could be considered. These options are reminiscent of Hoare, Smith and Egli-Milner powerdomains respectively [Plotkin 1981].

At this stage, the difference between don't know and don't care nondeterminism arises only in the way the *observables* are interpreted: in a don't care approach, agent  $A$  can commit to *one* of the final stores  $\sigma \Downarrow_{var(A)}$ , while, in a don't know approach, in classical cc programming it is enough that one of the final stores is consistent. Since existential quantification corresponds to the sum in our semiring-based approach, for us a don't know approach leads to the sum (that is, the lub) of all final stores:

$$S_A^{dk} = \bigoplus_{\sigma \in S_A} \sigma.$$

It is now interesting to notice that the thresholds appearing in the program are related also to the observable sets:

**PROPOSITION 6.4** (THRESHOLDS AND  $S_A$  (1)). *For each  $\psi$ , we have  $S_A \supseteq S_{cut_\psi(A)}$ .*

**PROOF.** By definition of cuts (Definition 6.1), we can modify the agents only by changing the thresholds with a new level, greater than the previous one. So, easily, we can only cut away some computations.  $\square$

**COROLLARY 6.5** (THRESHOLDS AND  $S_A^{dk}$  (1)). *For each  $\psi$ , we have  $S_A^{dk} \supseteq S_{cut_\psi(A)}^{dk}$ .*

**PROOF.** It follows from the definition of  $S_A^{dk}$  and from Proposition 6.4.  $\square$

**THEOREM 6.6** (THRESHOLDS AND  $S_A$  (2)). *Let  $\psi \sqsubseteq glb\{\sigma \in S_A\}$ . Then  $S_A = S_{cut_\psi(A)}$ .*

**PROOF.** By Proposition 6.4, we have  $S_A \subseteq S_{cut_\psi(A)}$ . Moreover, since  $\psi$  is lower than all  $\sigma$  in  $S_A$ , by Theorem 6.3 we have that all the computations are also in  $S_{cut_\psi(A)}$ . So, the statement follows.  $\square$

Notice that, thanks to Theorem 6.6 and to Proposition 6.4, whenever we have a lower bound  $\psi$  of the glb of the final solutions, we can use  $\psi$  as a threshold to eliminate some computations. Moreover, we can prove the following theorem:

**THEOREM 6.7.** *Let  $\sigma \in S_A$  and  $\sigma \notin S_{cut_\psi(A)}$ . Then we have  $\sigma \sqsubset \psi$ .*

**PROOF.** If  $\sigma \in S_A$  and  $\sigma \notin S_{cut_\psi(A)}$ , it means that the cut eliminates some computations. So, at some step we have changed the threshold of some tell or ask agent. In particular, since we know by Theorem 6.3 that when  $\psi \sqsubseteq \sigma$  we do not modify the computation, we need  $\psi \not\sqsubseteq \sigma$ . Moreover, since the tell and ask rules fail only if  $\sigma \sqsubset \psi$ , we easily have the statement of the theorem.  $\square$

The following theorem relates thresholds and  $S_A^{dk}$ .

THEOREM 6.8 (THRESHOLDS AND  $\mathcal{S}_A^{dk}$  (2)). *Let  $\Psi_A = \{\sigma \in \mathcal{S}_A \mid \exists \sigma' \in \mathcal{S}_A \text{ with } \sigma' \sqsupseteq \sigma\}$  (that is,  $\Psi_A$  is the set of “greatest” elements of  $\mathcal{S}_A$ ). Let also  $\psi \sqsubseteq \text{glb}\{\sigma \in \Psi_A\}$ . Then  $\mathcal{S}_A^{dk} = \mathcal{S}_{\text{cut}_\psi(A)}^{dk}$ .*

PROOF. Since we have  $a + b = b \iff a \leq b$ , we easily have  $\bigoplus_{\sigma \in \mathcal{S}_A} \sigma = \bigoplus_{\sigma \in \Psi_A} \sigma$ . Now, by following a reasoning similar to Theorem 6.6, by applying a cut with a threshold  $\psi \sqsubseteq \text{glb}\{\sigma \in \Psi_A\}$  we do not eliminate any computation. So we obtain  $\mathcal{S}_A^{dk} = \bigoplus_{\sigma \in \mathcal{S}_A} \sigma = \bigoplus_{\sigma \in \Psi_A} \sigma = \mathcal{S}_{\text{cut}_\psi(A)}^{dk}$   $\square$

LEMMA 6.9. *Given any constraint  $\psi$ , we have:*

$$\mathcal{S}_A^{dk} \sqsubseteq \psi + \mathcal{S}_{\text{cut}_\psi(A)}^{dk}.$$

PROOF. Let  $S$  be the set of all solutions; then  $\mathcal{S}_A^{dk} = \text{lub}(S)$  and  $\mathcal{S}_{\text{cut}_\psi(A)}^{dk} = \text{lub}(S_1)$  where  $S_1 \subseteq S$ . The solutions that have been eliminated by the cut  $\psi$  (that is all the  $\sigma \in S - S_1$ ) are all lower than  $\psi$  by Theorem 6.7. So, it easily follows that  $\mathcal{S}_A^{dk} \sqsubseteq \psi + \mathcal{S}_{\text{cut}_\psi(A)}^{dk}$ .  $\square$

THEOREM 6.10. *Given any constraint  $\psi$ , we have:*

$$\mathcal{S}_{\text{cut}_\psi(A)}^{dk} \sqsubseteq \mathcal{S}_A^{dk} \sqsubseteq \psi + \mathcal{S}_{\text{cut}_\psi(A)}^{dk}.$$

PROOF. From Corollary 6.5, we have  $\mathcal{S}_{\text{cut}_\psi(A)}^{dk} \sqsubseteq \mathcal{S}_A^{dk}$ . From Lemma 6.9 we have instead  $\mathcal{S}_A^{dk} \sqsubseteq \psi + \mathcal{S}_{\text{cut}_\psi(A)}^{dk}$ .  $\square$

This theorem suggests a way to cut useless computations while generating the observable  $\mathcal{S}_A^{dk}$  of an scc program  $P$  starting from agent  $A$ . A very naive way to obtain such an observable would be to first generate all final states, of the form  $\langle \text{success}, \sigma_i \rangle$ , and then compute their lub. An alternative, smarter way to compute this same observable would be to do the following. First we start executing the program as it is, and find a first solution, say  $\sigma_1$ . Then we restart the execution applying the cut level  $\sigma_1$ .

By Theorem 6.8, this new cut level cannot eliminate solutions which influence the computation of the observable: the only solutions it will cut are those that are lower than the one we already found, thus useless in terms of the computation of  $\mathcal{S}_A^{dk}$ .

In general, after having found solutions  $\sigma_1, \dots, \sigma_k$ , we restart execution with cut level  $\psi = \sigma_1 + \dots + \sigma_k$ . Again, this will not cut crucial solutions but only some that are lower than the sum of those already found. When the execution of the program terminates with no solution we can be sure that the cut level just used (which is the sum of all solutions found) is the desired observable (in fact, by Theorem 6.10 when  $\mathcal{S}_{\text{cut}_\psi(A)}^{dk} = \psi$  we necessarily have  $\mathcal{S}_{\text{cut}_\psi(A)}^{dk} = \mathcal{S}_A^{dk} = \psi$ ).

In a way, such an execution method resembles a branch & bound strategy, where the cut levels have the role of the bounds.

The following corollary is important to show the correctness of this approach.

COROLLARY 6.11. *Given any constraint  $\psi \sqsubseteq \mathcal{S}_A^{dk}$ , we have:*

$$\mathcal{S}_A^{dk} = \psi + \mathcal{S}_{\text{cut}_\psi(A)}^{dk}.$$

PROOF. It easily comes from Theorem 6.10.  $\square$

Table IV. Failure in the scc language

$\frac{\sigma \otimes c \sqsubset \phi}{\langle tell(c) \rightarrow_{\phi} A, \sigma \rangle \longrightarrow fail}$	(Tell <sub>1</sub> )
$\frac{(\sigma \otimes c) \Downarrow_{\emptyset} < a}{\langle tell(c) \rightarrow^a A, \sigma \rangle \longrightarrow fail}$	(Valued-tell <sub>1</sub> )
$\frac{\sigma \sqsubset \phi}{\langle ask(c) \rightarrow_{\phi} A, \sigma \rangle \longrightarrow fail}$	(Ask <sub>1</sub> )
$\frac{\sigma \Downarrow_{\emptyset} < a}{\langle ask(c) \rightarrow^a A, \sigma \rangle \longrightarrow fail}$	(Valued-ask <sub>1</sub> )
$\frac{\langle E_1, \sigma \rangle \longrightarrow fail, \langle E_2, \sigma \rangle \longrightarrow fail}{\langle E_1 + E_2, \sigma \rangle \longrightarrow fail}$	(Nondeterminism <sub>1</sub> )
$\frac{\langle A_1, \sigma \rangle \longrightarrow fail}{\langle A_1 \parallel A_2, \sigma \rangle \longrightarrow fail}$	(Parallelism <sub>1</sub> )
$\frac{\langle A_2 \parallel A_1, \sigma \rangle \longrightarrow fail}{\langle A_2 \parallel A_1, \sigma \rangle \longrightarrow fail}$	

Let us now use this corollary to prove the correctness of the whole procedure above.

Let  $\sigma_1$  be the first final state reached by agent  $A$ . By stopping the algorithm after one step, what we have to prove is  $S_A^{dk} = \sigma_1 + S_{cut_{\sigma_1}(A)}^{dk}$ . Since  $\sigma_1$  is for sure lower than  $S_A^{dk}$ , this is true by Corollary 6.11.

By applying this procedure iteratively, we will collect a superset  $\Psi'_A$  of  $\Psi_A = \{\sigma \in S_A \mid \exists \sigma' \in S_A \text{ with } \sigma' \sqsupseteq \sigma\}$  ( $\Psi'_A$  is a superset of  $\Psi_A$  because we could collect a final state  $\sigma_i$  before computing a final state  $\sigma_j \sqsupseteq \sigma_i$ ; in this case both will be in  $\Psi'_A$ ). Even if  $\Psi'_A$  contains more elements than  $\Psi_A$ , we have  $\bigoplus_{\sigma \in \Psi'_A} = \bigoplus_{\sigma \in \Psi_A}$  (for the extensivity and idempotence properties of  $+$ ).

The only difference with the procedure we have tested correct w.r.t. the algorithm is that, at each step, it performs a cut by using the sum of all the previously computed final state. This means that the algorithm can at each step eliminate more computations, but by the results of Theorem 6.7 the eliminated computations does not change the final result.

## 6.2 Failure

The transition system we have defined considers only successful computations. If this could be a reasonable choice in a don't know interpretation of the language it will lead to an insufficient analysis of the behaviour in a *pessimistic* interpretation of the indeterminism. To capture agents' failure, we add the terminal *fail* to the configurations and the transition rules of Table IV to those of Table III.

(*Valued*)*tell*<sub>1</sub>/*ask*<sub>1</sub>. The failing rule for ask and tell simply checks if the added/checked constraint  $c$  is *inconsistent* with the store  $\sigma$  and in this case stops the computation and gives *fail* as a result. Note that since we use soft constraints we enriched this operator with a threshold ( $a$  or  $\phi$ ). This is used also to compute failure. If the level of consistency of the resulting store is lower than the threshold level, then this is considered a failure.

Table V. Hanging computations in the scc language

$\sigma \not\models c, \sigma \not\models \phi$	
$\frac{}{\langle ask(c) \rightarrow_{\phi} A, \sigma \rangle \longrightarrow hang}$	(Ask <sub>2</sub> )
$\sigma \not\models c, \sigma \Downarrow_{\emptyset} \not\models a$	
$\frac{}{\langle ask(c) \rightarrow^a A, \sigma \rangle \longrightarrow hang}$	(Valued-ask <sub>2</sub> )
$\frac{\langle E_1, \sigma \rangle \longrightarrow fail/hang, \langle E_2, \sigma \rangle \longrightarrow hang}{\langle E_1 + E_2, \sigma \rangle \longrightarrow hang}$	(Nondeterminism <sub>2</sub> )
$\frac{}{\langle E_2 + E_1, \sigma \rangle \longrightarrow hang}$	

*Nondeterminism<sub>1</sub>*. Since the failure of a branch arises only from the failure of a guard, and since we use angelic non-determinism (that is, we check the guards before choosing one path), we fail only when all the branches fail.

*Parallelism<sub>1</sub>*. In this case the computation fails as soon as one of the branches ails.

The observables of each agent can now be enlarged by using the function

$$\mathcal{F}_A = \{fail \mid \langle A, \mathbf{1}_V \rangle \rightarrow^* fail\}$$

that computes a failure if at least a computation of agent  $A$  fails.

By considering also the failing computations, the difference between don't know and don't care becomes finer. In fact, in situations where we have  $\mathcal{S}_A = \mathcal{S}_A^{dk}$ , the failing computations could make the difference: in the don't care approach the notion of failure is *existential* and in the don't know one becomes *universal* [de Boer and Palamidessi 1994]:

$$\mathcal{F}_A^{dk} = \{fail \mid \text{all computations for } A \text{ which lead to } fail\}.$$

This means that in the don't know nondeterminism we are interested in observing a failure only if all the branches fail. In this way, given an agent  $A$  with an empty  $\mathcal{S}_A^{dk}$  and a non-empty  $\mathcal{F}_A^{dk}$ , we cannot say for sure that the semantic of this agent is *fail*. In fact, the transition rules we have defined do not consider *hang* and infinite computations. Similar semidecibility results for soft constraint logic programming are proven in [Bistarelli et al. 2001].

### 6.3 Hanging and infinite computations

To complete the possible observables of a goal, we need also to observe the hanged states or those representing infinite computation. To this extent, we extend the configurations with the terminals *hang* and  $\perp$ , and we add some transition rules (those in Table V) to handle hanging computations.

*Nondeterminism*. The only case that can lead the system to a hanging state is when all the branches are stuck. In this case, we can assume no future change of the state will happen that give the possibility to the agents to evolve.

To deal with hang states and infinite computations, we enlarged the observables with the functions

$$\mathcal{H}_A = \{hang \mid \langle A, \mathbf{1} \rangle \rightarrow^* hang\}$$

that collects all the hang computations of the agent  $A$  and

$$\mathcal{D}_A = \{\perp \mid \langle A, \mathbf{1} \rangle \text{ diverges} \}$$

to represent infinite computations.

## 7. AN EXAMPLE FROM THE NETWORK SCENARIO

We consider in this section a simple network problem, involving a set of processes running on distinct locations and sharing some variables, over which they need to synchronize, and we show how to model and solve such a problem in scc.

Each process is connected to a set of variables, shared with other processes, and it can perform several moves. Each of such moves involves performing an action over some or all the variables connected to the process. An action over a variable consists of giving a certain value to that variable. A special value “idle” models the fact that a process does not perform any action over a variable. Each process has also the possibility of not moving at all: in this case, all its variables are given the idle value.

The desired behavior of a network of such processes is that, at each move of the entire network:

- (1) processes sharing a variable perform the same action over it;
- (2) as few processes as possible remain idle.

To describe a network of processes with these features, we use an SCSP where each variable models a shared variable, and each constraint models a process and connects the variables corresponding to the shared variables of that process. The domain of each variable in this SCSP is the set of all possible actions, including the idle one. Each way of satisfying a constraint is therefore a tuple of actions that a process can perform on the corresponding shared variables.

In this scenario, softness can be introduced both in the domains and in the constraints. In particular, since we prefer to have as many moving processes as possible, we can associate a penalty to both the idle element in the domains, and to tuples containing the idle action in the constraints. As for the other domain elements and constraint tuples, we can assign them suitable preference values to model how much we like that action or that process move.

For example, we can use the semiring  $S = \langle [-\infty, 0], \max, +, -\infty, 0 \rangle$ , where 0 is the best preference level (or, said dually, the weakest penalty),  $-\infty$  is the worst level, and preferences (or penalties) are combined by summing them. According to this semiring, we can assign value  $-\infty$  to the idle action or move, and suitable other preference levels to the other values and moves. Figure 2 gives the details of a part of a network and it shows eight processes (that is,  $c_1, \dots, c_8$ ) sharing a total of six variables. In this example, we assume that processes  $c_1, c_2$  and  $c_3$  are located on site  $a$ , processes  $c_5$  and  $c_6$  are located on site  $b$ , and  $c_4$  is located on site  $c$ . Processes  $c_7$  and  $c_8$  are located on site  $d$ . Site  $e$  connects this part of the network to the rest. Therefore, for example, variables  $x_d, y_d$  and  $z_d$  are shared between processes located in distinct locations.

As desired, finding the best solution for the SCSP representing the current state of the process network means finding a move for all the processes such that they

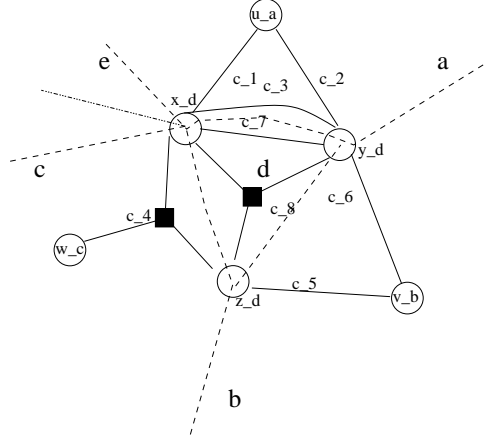
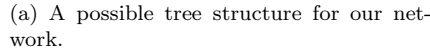


Fig. 2. The SCSP describing part of a process network

perform the same action on the shared variables and there is a minimum number of idle processes. However, since the problem is inherently distributed, it does not make sense, and it might not even be possible, to centralize all the information and give it to a single soft constraint solver.

On the contrary, it may be more reasonable to use several soft constraint solvers, one for each network location, which will take care of handling only the constraints present in that location. Then, the interaction between processes in different locations, and the necessary agreement to solve the entire problem, will be modelled via the scc framework, where each agent will represent the behaviour of the processes in one location.

More precisely, each scc agent (and underlying soft constraint solver) will be in charge of receiving the necessary information from the other agents (via suitable asks) and using it to achieve the synchronization of the processes in its location. For this protocol to work, that is, for obtaining a global optimal solution without a centralization of the work, the SCSP describing the network of processes has to have a tree-like shape, where each node of the tree contains all the processes in a location, and the agents have to communicate from the bottom of the tree to its root. In fact, the proposed protocol uses a sort of Dynamic Programming technique to distribute the computation between the locations. In this case the use of a tree shape allows us to work, at each step of the algorithm, only locally to one of the locations. In fact, a non tree shape would lead to the construction of non-local constraints and thus require computations which involve more than one location at a time. In our example, the tree structure we will use is the one shown in Figure 3(a), which also shows the direction of the child-parent relation links (via arrows). Figure 3(b) describes instead the partition of the SCSP over the four involved locations. The gray connections represent the synchronization to be assured between distinct locations. Notice that, w.r.t. Figure 2, we have duplicated the variables representing variables shared between distinct locations, because of our desire to first perform a local work and then to communicate the results to the other locations.



(b) The SCSP partitioned over the four locations.

The scc agents (one for each location plus the parallel composition of all of them) are therefore defined as follows:

$$\begin{aligned}
A_a &: \exists u_a (tell(c_1(x_a, u_a) \wedge c_2(u_a, y_a) \wedge c_3(x_a, y_a)) \rightarrow tell(end_a = true) \rightarrow stop) \\
A_b &: \exists v_b (tell(c_5(y_b, v_b) \wedge c_6(z_b, v_b)) \rightarrow tell(end_b = true) \rightarrow stop) \\
A_c &: \exists w_c (tell(c_4(x_c, w_c, z_c)) \rightarrow tell(end_c = true) \rightarrow stop) \\
A_d &: ask(end_a = true \wedge end_b = true \wedge end_c = true \wedge end_d = true) \rightarrow \\
&\quad tell(c_7(x_d, y_d) \wedge c_8(x_d, y_d, z_d) \wedge x_a = x_d = x_c \wedge y_a = y_d = y_b \wedge z_b = z_d = z_c) \\
&\quad \rightarrow tell(end_d = true) \rightarrow stop \\
A &: A_a \mid A_b \mid A_c \mid A_d
\end{aligned}$$

Agents  $A_a, A_b, A_c$  and  $A_d$  represent the processes running respectively in the location  $a, b, c$  and  $d$ . Note that, at each ask or tell, the underlying soft constraint solver will only check (for consistency or entailment) a part of the current set of constraints: those local to one location. Due to the tree structure chosen for this example, where agents  $A_a, A_b$ , and  $A_c$  correspond to leaf locations, only agent  $A_d$  shows all the actions of a generic process: first it needs to collect the results computed separately by the other agents (via the ask); then it performs its own constraint solving (via a tell), and finally it can set its end flag, that will be used by a parent agent (in this case the agent corresponding to location  $e$ , which we have not modelled here).

## 8. CONCLUSIONS AND FUTURE WORK

We have shown that cc languages can deal with soft constraints. Moreover, we have extended their syntax to use soft constraints also to direct and prune the search process at the language level. We believe that such a new programming paradigm could be very useful for web and internet programming.

In fact, in several network-related areas, constraints are already being used [Bella and Bistarelli 2001; Awduche et al. 1999; Clark 1989; Jain and Sun 2000; Calisti and Faltings 2000]. The soft constraint framework has the advantage over the classical one of selecting a “best” solution also in overconstrained or underconstrained systems. Moreover, the need to express preferences and to search for optimal solutions shows that soft constraints can improve the modelling of web interaction scenarios.

## ACKNOWLEDGMENTS

We are indebted to Paolo Baldan for valuable suggestions.

## REFERENCES

- AWDUCHE, D., MALCOLM, J., AGOGBUA, J., O'DELL, M., AND MCMANUS, J. 1999. RFC2702: Requirements for traffic engineering over mpls. Tech. rep., Network Working Group. Sept.
- BELLA, G. AND BISTARELLI, S. 2001. Soft constraints for security protocol analysis: Confidentiality. In *Proceedings of the 3rd International Symposium on Practical Aspects of Declarative Languages (PADL '01)*, I. Ramakrishnan, Ed. LNCS, vol. 1990. Springer-Verlag, Heidelberg, Germany, 108–122.
- BISTARELLI, S. 2001. Soft constraint solving and programming: a general framework. Ph.D. thesis, Dipartimento di Informatica, Università di Pisa, Italy. TD-2/01.
- BISTARELLI, S., CODOGNET, P., AND ROSSI, F. 2000b. Abstracting soft constraints. In *Proceedings of the 1999 ERCIM/Compulog Net workshop on Constraints*, K. Apt, E. Monfroy, T. Kakas, and F. Rossi, Eds. LNCS, vol. 1865. Springer, Heidelberg, Germany.
- BISTARELLI, S., CODOGNET, P., AND ROSSI, F. 2000a. An abstraction framework for soft constraints and its relationship with constraint propagation. In *Proceedings of the Symposium on Abstraction, Reformulation and Approximation (SARA2000)*, B. Y. Chouery and T. Walsh, Eds. LNAI, vol. 1864. Springer, Heidelberg, Germany.
- BISTARELLI, S., MONTANARI, U., AND ROSSI, F. 1995. Constraint Solving over Semirings. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI'95)*. Morgan Kaufman, San Francisco, CA, USA.
- BISTARELLI, S., MONTANARI, U., AND ROSSI, F. 1997. Semiring-based Constraint Solving and Optimization. *Journal of the ACM* 44, 2 (Mar), 201–236.
- BISTARELLI, S., MONTANARI, U., AND ROSSI, F. 2001. Semiring-based Constraint Logic Programming: Syntax and Semantics. *ACM Trans. Program. Lang. Syst.* 23, 1–29.
- CALISTI, M. AND FALTINGS, B. 2000. Distributed constrained agents for allocating service demands in multi-provider networks. *Journal of the Italian Operational Research Society XXIX*, 91. Special Issue on Constraint-Based Problem Solving.
- CHEN, S. AND NAHRSTEDT, K. 1998. Distributed QoS routing with imprecise state information. In *ICCCCN98*.
- CLARK, D. 1989. RFC1102: Policy routing in internet protocols. Tech. rep., Network Working Group. May.
- DE BOER, F. AND PALAMIDESSI, C. 1991. A fully abstract model for concurrent constraint programming. In *Proceeding of the 16th Colloquium on Trees in Algebra and Programming (CAAP1991)*, S. Abramsky and T. Maibaum, Eds. Vol. 493. Springer-Verlag, Heidelberg, Germany.



- DE BOER, F. AND PALAMIDESSI, C. 1994. From Concurrent Logic Programming to Concurrent Constraint Programming. In *Advances in Logic Programming Theory*, G. Levi, Ed. Oxford University Press, 55–113.
- DUBOIS, D., FARGIER, H., AND PRADE, H. 1993. The calculus of fuzzy restrictions as a basis for flexible constraint satisfaction. In *Proceedings of the 2nd IEEE International Conference on Fuzzy Systems (FUZZ-IEEE 1993)*. IEEE, Piscataway, NJ, U.S.A., 1131–1136.
- FARGIER, H. AND LANG, J. 1993. Uncertainty in constraint satisfaction problems: a probabilistic approach. In *Proceeding of the 2nd European Conference on Symbolic and Quantitative Approaches to Reasoning with Uncertainty (ECSQARU1993)*. LNCS, vol. 747. Springer-Verlag, Heidelberg, Germany, 97–104.
- FREUDER, E. AND WALLACE, R. 1992. Partial constraint satisfaction. *Artificial Intelligence Journal* 58.
- JAIN, R. AND SUN, W. 2000. QoS/Policy/Constraint-based routing. In *Carrier IP Telephony 2000 Comprehensive Report*. International Engineering Consortium, Heidelberg, Germany. ISBN: 0-933217-75-7.
- PLOTKIN, G. 1981. Post-graduate lecture notes in advanced domain theory (incorporating the pisa lecture notes). Technical report, Dept. of Computer Science, Univ. of Edinburgh.
- RUTTKAY, Z. 1994. Fuzzy constraint satisfaction. In *Proceedings of the 3rd IEEE International Conference on Fuzzy Systems (FUZZ-IEEE 1994)*. 1263–1268.
- SARASWAT, V. 1993. *Concurrent Constraint Programming*. MIT Press.
- SCHIEX, T. 1992. Possibilistic constraint satisfaction problems, or “how to handle soft constraints?”. In *Proceeding of the 8th Conference on Uncertainty in Artificial Intelligence (UAI1992)*. 269–275.
- SCHIEX, T., FARGIER, H., AND VERFAILLE, G. 1995. Valued Constraint Satisfaction Problems: Hard and Easy Problems. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI’95)*. Morgan Kaufmann, San Francisco, CA, USA, 631–637.

...