Soft Concurrent Constraint Programming

Stefano Bistarelli
Istituto di Informatica e Telematica, C.N.R., Pisa, Italy
Dipartimento di Scienze, Universitá degli Studi "G. D'Annunzio", Pescara, 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.

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

The main idea is to choose a *constraint system* and use constraints to model communication 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 security protocol [Bella and Bistarelli 2001; 2002] and integrity policy analysis [Bistarelli and Foley 2003a; 2003b] have shown to be enhanced by using security levels instead of a simple notion of secure/insecure level . 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 first compare the semiring-based framework with constraint systems *"a la Saraswat"* and then we show how use it inside the cc framework. More precisely, we describe how to use soft constraints instead of classical ones within the original cc framework. In this scenario, the only addition to classical cc is the use of a function which transforms preference levels into a yes/no information of consistency.

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 determine their success, failure, or suspension, as well as 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) and *fail* (§6.2) computations 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 [Scott 1982]) 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 $\wp(D) \times \wp(D))$[1] where $\wp(D)$ is the powerset of D. The entailment relation satisfies:

—$u \vdash P$ for all $P \in u$ (reflexivity) and

—if $u \vdash v$ and $v \vdash z$, then $u \vdash z$ for all $u, v, z \in \wp(D)$ (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 [Saraswat et al. 1991], $\langle \overline{\wp(D)}, \subseteq\rangle$ 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 $\overline{\wp(D)}$; $c \supseteq d$ means $c \vdash d$.

2.1.2 *The Hiding Operator: Cylindric Algebras.* In order to treat the hiding operator of the language (see Definition 3.10), 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 $x \in V$, $\exists_x : \overline{\wp(D)} \to \overline{\wp(D)}$ 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}. If $\vdash$ models the equality theory, then the elements d_{xy} can be thought of as the formulas $x = y$. Such a constraint 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 $d_{xy} \sqcup \exists_x(c \sqcup d_{xy}) \vdash c$.

[1]The extension is s.t. $u \vdash v$ iff $u \vdash P$ for every $P \in v$.

Table I. cc syntax

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

Note that the in the previous definition we assume the cardinality of the domain for x, y and z greater than 1 (otherwise, axioms 2 and 3 would not make sense).

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 $vars(A)$ 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 "$\sum_{i:=1,n} ask(c_i) \rightarrow A_i$" behaves as follows: if there is at least one c_i which is entailed by the current store, it behaves like A_i; if all c_i are inconsistent with the current store, it fails; otherwise it suspends.

—agent "$tell(c) \rightarrow A$" adds constraint c to the current store and then, if the resulting store is consistent, behaves like A, otherwise it fails.

—agent $A_1 \| A_2$ 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; 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 $\langle \mathcal{A}, +, \times, \mathbf{0}, \mathbf{1} \rangle$ such that:

(1) $\mathcal{A}$ is a set and $\mathbf{0}, \mathbf{1} \in \mathcal{A}$;
(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*[2] is a semiring $\langle \mathcal{A}, +, \times, \mathbf{0}, \mathbf{1} \rangle$ such that $+$ is idempotent, $\mathbf{1}$ is its absorbing element and $\times$ is commutative. Let us consider the relation $\leq_S$ over $\mathcal{A}$ 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) $\times$ is intensive on $\leq_S$: $a \times b \leq_S a, b$;
(4) $\mathbf{0}$ is its minimum and $\mathbf{1}$ its maximum;
(5) $\langle \mathcal{A}, \leq_S \rangle$ is a complete lattice and, for all $a, b \in \mathcal{A}$, $+$ is the least upper bound operator, that is, $a + b = lub(a, b)$.

Moreover, if $\times$ is idempotent, then: $+$ distributes over $\times$; $\langle \mathcal{A}, \leq_S \rangle$ 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 $S = \langle \mathcal{A}, +, \times, \mathbf{0}, \mathbf{1} \rangle$, a set D (the domain of the variables) and an ordered set of variables V, a *constraint* is a pair $\langle def, con \rangle$ where $con \subseteq V$ and $def : D^{|con|} \rightarrow \mathcal{A}$. Therefore, a constraint specifies a set of variables (the ones in con), and assigns to each tuple of values of these variables an element of the semiring. Consider two constraints $c_1 = \langle def_1, con \rangle$ and $c_2 = \langle def_2, con \rangle$, with $|con| = k$. Then $c_1 \sqsubseteq_S c_2$ if for all k-tuples t, $def_1(t) \leq_S def_2(t)$. The relation $\sqsubseteq_S$ is a partial order.

A *soft constraint problem* is a pair $\langle C, con \rangle$ where $con \subseteq V$ and C is a set of constraints: con is the set of variables of interest for the constraint set C, which however may concern also variables not in con. Note that a classical CSP is a SCSP where the chosen c-semiring is: $S_{CSP} = \langle \{false, true\}, \vee, \wedge, false, true \rangle$. In this case, the $\sqsubseteq$ relation reduces to set inclusion between the sets of allowed tuples.

Fuzzy CSPs [Schiex 1992] can instead be modeled in the SCSP framework by choosing the c-semiring $S_{FCSP} = \langle [0, 1], max, min, 0, 1 \rangle$. Many other "soft" CSPs (Probabilistic, weighted, ...) can be modeled by using a suitable semiring structure (for example, $S_{prob} = \langle [0, 1], max, \times, 0, 1 \rangle$, $S_{weight} = \langle \mathcal{R}, min, +, 0, +\infty \rangle$, ...).

Since the Cartesian product of two c-semirings is still a c-semiring [Bistarelli et al. 1997], it is also possible to model multicriteria optimization within this framework.

[2] "c" stands for "constraint".

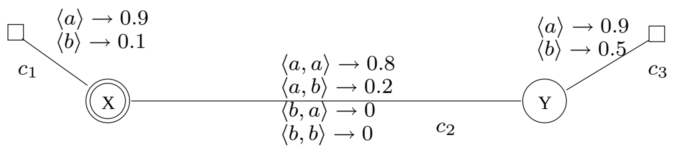


Fig. 1. A fuzzy CSP

For example we can model situations where we want to maximize the minimum preference and also to minimize the sum of the costs. To do this we just have to pair the fuzzy and the weighted semiring.

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 *con*) are represented with a double circle. Here we assume that the domain D of the variables contains only elements a and b.

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 $def(t) = def_1(t \downarrow^{con}_{con_1}) \times def_2(t \downarrow^{con}_{con_2})$, where $t \downarrow^X_Y$ 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 $def'(t') = \sum_{t \text{ s.t. } t\downarrow^{con}_{I\cap con}=t'} def(t)$. 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 0.8. 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). Notice that $blevel(P) = \langle \emptyset \rightarrow \alpha, \emptyset\rangle$ is a constraint with empty *con* (that is, no variable is involved) and the function def is simply a constant represented by the semiring value α. We also say that: P is α-consistent if $blevel(P) = \alpha$; P is consistent iff there exists $\alpha > \mathbf{0}$ such that P is α-consistent; P is inconsistent if it is not consistent.

3. CONCURRENT CONSTRAINT PROGRAMMING OVER SOFT CONSTRAINTS

Given a semiring $S = \langle \mathcal{A}, +, \times, \mathbf{0}, \mathbf{1}\rangle$ and an ordered set of variables V over a 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 $\mathcal{C} = (V \rightarrow D) \rightarrow \mathcal{A}$ as the set of all possible constraints that can be built starting from $S = \langle \mathcal{A}, +, \times, \mathbf{0}, \mathbf{1}\rangle$, 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 η 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 $supp(c) = \{v \in V \mid \exists \eta, d_1, d_2 . c\eta[v := d_1] \neq c\eta[v := d_2]\}$, 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 η' is η modified with the association $v := d_1$ (that is the operator [ ] has precedence over application).

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

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:

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

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 η, 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 $supp(c \Downarrow_I) \subseteq I$).*

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 $x \notin supp(c \Downarrow_I)$. □

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

—*$\mathcal{C} : (V \to D) \to \mathcal{A}$ 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:
$(c_1 \otimes c_2)\eta =$ (by definition of $\otimes$)
$c_1\eta \times c_2\eta =$ (by commutativity of $\times$)
$c_2\eta \times c_1\eta =$ (by definition of $\otimes$)
$(c_2 \otimes c_1)\eta$.
All the other properties can be proved similarly. □

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 higher-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 intensivity 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. $C_1 \vdash C_2 \iff C_1 \vdash \bigotimes C_2$. Note that when $\times$ is idempotent we have that, $\forall c_2 \in C_2,\ C_1 \vdash c_2 \iff C_1 \vdash \bigotimes C_2$. In this case to prove the item we have to prove that if $\bigotimes C_1 \sqsubseteq \bigotimes C_2$ and $\bigotimes C_2 \sqsubseteq \bigotimes C_3$, then $\bigotimes C_1 \sqsubseteq \bigotimes C_3$. 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 $\bigotimes C_1$. This set is univocally representable by the constraint function $\bigotimes C_1$.

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 the algebricity, since the algebraic nature of the structure $\mathcal{C}$ strictly depends on the properties of the semiring.

Notice that we do not aim at computing the closure of the entailment relation, but only to use the entailment relation to establish if a constraint is entailed by the current store, and this can be established even if the lattice is not algebraic.

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 $(\exists_x c)\eta = \sum_{d_i \in D} c\eta[x := d_i]$.

To make the hiding operator computationally tractable, we require that the number of domain elements in D having semiring value different from $\mathbf{0}$ is finite. In this way, to compute the sum needed for $(\exists_x c)\eta$ in the above definition, we can consider just a finite number of elements (those different from $\mathbf{0}$) since $\mathbf{0}$ is the unit element of the sum. The same result can also be achieved by imposing some other restriction on the constraints.

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

Proposition 3.11. *Consider a semiring $S = \langle \mathcal{A}, +, \times, \mathbf{0}, \mathbf{1} \rangle$, a domain of the variables D, an ordered set of variables V, the corresponding structure $\mathcal{C}$ and the class of hiding functions $\exists_x : \mathcal{C} \to \mathcal{C}$ 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 $\sum_{d_i \in D} c\eta[x := d_i]$. □

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

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

Theorem 3.12. *Consider a semiring $S = \langle \mathcal{A}, +, \times, \mathbf{0}, \mathbf{1} \rangle$, a domain of the variables D, an ordered set of variables V, the corresponding structure $\mathcal{C}$ and the class of hiding functions $\exists_x : \mathcal{C} \to \mathcal{C}$ 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) $\exists_x (c_1 \otimes \exists_x c_2) =$
$(c_1 \otimes \exists_x c_2) \Downarrow_{V-x} =$
$(c_1 \otimes c_2 \Downarrow_{V-x}) \Downarrow_{V-x}$ (since $con(c_2 \Downarrow_{V-x}) = V - x$, and $V - x \cap x = \emptyset$, from Theorem 19 of [Bistarelli et al. 1997] this is equivalent to)
$c_1 \Downarrow_{V-x} \otimes c_2 \Downarrow_{V-x} =$
$\exists_x c_1 \otimes \exists_x c_2$;

(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 $supp(d_{xy}) = \{x, y\}$.

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

THEOREM 3.14. *Consider a semiring $S = \langle \mathcal{A}, +, \times, \mathbf{0}, \mathbf{1} \rangle$, 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. So, since for any c, $\mathbf{0} \vdash c$ and $c \vdash c$, we easily have the claim of the theorem.

□

3.1 Using cc on Top of a Soft Constraint System

When using a soft constraint system in a cc language, the notion of consistency should be generalised. In fact, SCSPs with best level of consistency equal to $\mathbf{0}$ can be interpreted as inconsistent, and those with level greater than $\mathbf{0}$ as consistent, but we can also be more general: we can define a suitable function α that, given the best level of the current store, maps such a level over the classical notion of consistency/inconsistency.

More precisely, given a semiring $S = \langle \mathcal{A}, +, \times, \mathbf{0}, \mathbf{1} \rangle$, we can define a function $\alpha : \mathcal{A} \rightarrow \{false, true\}$. Function α has to be at least monotone, but functions with a richer set of properties could be used. 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 α over $true$ or $false$.

It is important to notice that changing the α function (that is, by mapping in a different way the set of values $\mathcal{A}$ 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 $\langle S, D, V \rangle$, 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

Table II. scc syntax

$P ::= F.A$

$F ::= p(X) :: A \mid F.F$

$A ::= success \mid fail \mid tell(c) \rightarrow_{\phi} A \mid tell(c) \rightarrow^{a} A \mid E \mid A \| A \mid \exists X.A \mid p(X)$

$E ::= ask(c) \rightarrow_{\phi} A \mid ask(c) \rightarrow^{a} A \mid E + E$

of a semiring element a and of a constraint ϕ to be checked whenever an *ask* or *tell* operation is performed. More precisely, the level a (resp., ϕ) will be used as a cut level to prune computations that are not good enough.

Notice also the we add the *fail* state but the failure Semantics will be considered in Section 6.2.

We present here a structured operational semantics for scc programs, in the SOS style, which consists of defining the semantics of the programming language by specifying a set of *configurations* Γ, 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, 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 Γ 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 $\Gamma = \{\langle A, \sigma \rangle\}\}$, 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:

Valued-tell. The valued-tell rule checks for the α-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 $b \not< a$[3]. 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 ϕ 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 ϕ. Notice that this notion of tell could be also applied to the classical cc framework. In this case the tell operation would succeed when the set of tuples satisfying constraint ϕ is a subset of the set of tuples allowed by $\sigma \cap c$.[4]

[3]Notice that we use $b \not< a$ instead of $b \geq a$ because we can possibly deal with partial orders. The same happens also in other transition rules with $\not\sqsubset$ instead of $\sqsupseteq$.

[4]notice that the $\otimes$ operator in the crisp case reduces to set intersection.

Table III. Transition rules for scc

$$\frac{(\sigma \otimes c) \Downarrow_{\emptyset} \not< a}{\langle tell(c) \rightarrow^{a} A, \sigma \rangle \longrightarrow \langle A, \sigma \otimes c \rangle} \qquad \text{(Valued-tell)}$$

$$\frac{\sigma \otimes c \not\sqsubseteq \phi}{\langle tell(c) \rightarrow_{\phi} A, \sigma \rangle \longrightarrow \langle A, \sigma \otimes c \rangle} \qquad \text{(Tell)}$$

$$\frac{\sigma \vdash c, \sigma \Downarrow_{\emptyset} \not< a}{\langle ask(c) \rightarrow^{a} A, \sigma \rangle \longrightarrow \langle A, \sigma \rangle} \qquad \text{(Valued-ask)}$$

$$\frac{\sigma \vdash c, \sigma \not\sqsubseteq \phi}{\langle ask(c) \rightarrow_{\phi} A, \sigma \rangle \longrightarrow \langle A, \sigma \rangle} \qquad \text{(Ask)}$$

$$\frac{\langle A_1, \sigma \rangle \longrightarrow \langle A'_1, \sigma' \rangle}{\begin{array}{c}\langle A_1 \| A_2, \sigma \rangle \longrightarrow \langle A'_1 \| A_2, \sigma' \rangle \\ \langle A_2 \| A_1, \sigma \rangle \longrightarrow \langle A_2 \| A'_1, \sigma' \rangle\end{array}} \qquad \frac{\langle A_1, \sigma \rangle \longrightarrow \langle success, \sigma' \rangle}{\begin{array}{c}\langle A_1 \| A_2, \sigma \rangle \longrightarrow \langle A_2, \sigma' \rangle \\ \langle A_2 \| A_1, \sigma \rangle \longrightarrow \langle A_2, \sigma' \rangle\end{array}}$$
$$\text{(Parallelism)}$$

$$\frac{\langle E_1, \sigma \rangle \longrightarrow \langle A_1, \sigma' \rangle}{\begin{array}{c}\langle E_1 + E_2, \sigma \rangle \longrightarrow \langle A_1, \sigma' \rangle \\ \langle E_2 + E_1, \sigma \rangle \longrightarrow \langle A_1, \sigma' \rangle\end{array}} \qquad \text{(Nondeterminism)}$$

$$\frac{\langle A[y/x], \sigma \rangle \longrightarrow \langle A', \sigma' \rangle}{\langle \exists_x A, \sigma \rangle \longrightarrow \langle A', \sigma' \rangle} \text{ with } y \textit{ fresh} \qquad \text{(Hidden variables)}$$

$$\langle p(y), \sigma \rangle \longrightarrow \langle A[y/x], \sigma \rangle \text{ when } p(x) :: A \qquad \text{(Procedure call)}$$

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 σ 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 ϕ is compared to the store σ. Notice that we need to check $\sigma \not\sqsubseteq \phi$ because previous tells could have a different threshold ϕ' and could not guarantee the consistency of the resulting store.

Nondeterminism and parallelism. The composition operators $+$ and $\|$ 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: as usual, we use the notion of diagonal constraints (as defined in Definition 3.13) to model 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:

$$\langle tell(c) \rightarrow A, \sigma\rangle \longrightarrow \langle A, \sigma \otimes c\rangle \qquad \text{(Eventual tell)}$$

$$\frac{\sigma \vdash c}{\langle ask(c) \rightarrow A, \sigma\rangle \longrightarrow \langle A, \sigma\rangle} \qquad \text{(Eventual ask)}$$

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 α-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

$$c : \{x, y\} \rightarrow \mathcal{R}^2 \rightarrow [0, 1] \qquad \text{s.t. } c(x, y) = \frac{1}{1 + |x - y|} \qquad \text{and}$$

$$c' : \{x\} \rightarrow \mathcal{R} \rightarrow [0, 1] \qquad \text{s.t. } c'(x) = \begin{cases} 1 & \text{if } x \leq 10, \\ 0 & \text{otherwise.} \end{cases}$$

Notice that the domain of both variables x and y is in this example any integer (or real) number. As any fuzzy CSP, the codomain of the constraints is instead in the interval $[0, 1]$[5].

Let's now evaluate the agent

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

in the empty starting store 1. Note that also here the empty store 1 is just the store containing the constraint $\langle \emptyset \rightarrow 1, \emptyset\rangle$ with empty support and that assign always the semiring level 1 to any assignment.

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

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

Now we need to check (by following the rule of *Valued-ask*) if $c \vdash c'$ and $c \Downarrow_{\emptyset} \not< 0.8$. 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

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

[5]In this case the number of domain elements in D having semiring value different from $\mathbf{0}$ is not finite; however, since we do not use the hiding operator we have no computational tractability problem.

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

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

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 ϕ, in any subagent of A or in any clause of the program, are replaced by ψ if $\phi \sqsubseteq \psi$. This means that the cut level of each subagent and clause becomes at least ψ, or is left to the original level. Informally, using the cut ψ we want to obtain (if possible) a solution not lower than ψ, so, all the ask/tell checks have to be increased in order to cut away computation with a store not better than ψ. Similar cuts can be also done using semiring levels a instead of constraints ψ.

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

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

$$cut^a(ask/tell(c) \rightarrow^{a'}) = \begin{cases} ask/tell(c) \rightarrow^a & \text{if } a' \leq a, \\ ask/tell(c) \rightarrow^{a'} & \text{otherwise.} \end{cases}$$

By definition, it is easy to see that $cut_{\mathbf{0}}(A) = cut^0(A) = A$.

We can then prove the following Lemma (some of them will be useful later):

LEMMA 6.2 (TELL AND ASK CUT). *Consider the Tell and Ask rules of Table III, and the constraints σ 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 $\psi \sqsubseteq \sigma$.*

—*If the* Valued-tell *rule can be applied to agent A, then the rule can be applied also to $cut^a(A)$ when $a \leq (\sigma \otimes c) \Downarrow_\emptyset$.*

—*If the* Valued-ask *rule can be applied to agent A, then the rule can be applied also to $cut^a(A)$ when $a \leq \sigma \Downarrow_\emptyset$.*

PROOF. We will prove only the first item; the others 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 σ is the store we have $\sigma \otimes c \not\sqsubset \phi$. Now, by definition of cut_ψ, we can have

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

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

In the first case, by hypothesis we have $\sigma \otimes c \not\sqsubseteq \phi$, which together with $\phi \sqsubset \psi$ implies $\sigma \otimes c \not\sqsubseteq \phi$, which is the required condition for the application of the Tell rule. In the second case, the statement directly holds by the hypothesis over A that $\sigma \otimes c \not\sqsubseteq \phi$. $\square$

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*

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

for a program P, and let $a = \sigma \Downarrow_{\emptyset}$. Then, also

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

and

$$\langle cut^{a}(A), \mathbf{1}\rangle \rightarrow \langle cut^{a}(A_1), \sigma_1\rangle \rightarrow \ldots \langle cut^{a}(A_n), \sigma_n\rangle \rightarrow \langle success, \sigma\rangle \qquad (2)$$

are scc computations 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 intensive, the store can only monotonically decrease starting from the initial store $\mathbf{1}$ and ending in the final store σ. So we have

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

Notice also that we have

$$1 \leq \sigma_1 \Downarrow_{\emptyset} \ldots \leq \sigma_n \Downarrow_{\emptyset} \leq \sigma \Downarrow_{\emptyset} = a.$$

We will prove (1) ((2) can be proven in the same way). 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 σ 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 σ 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:

$$\mathcal{S}_A^{dk} = \bigoplus_{\sigma \in \mathcal{S}_A} \sigma.$$

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 $\mathcal{S}_A$ (1)). *For each ψ, we have $\mathcal{S}_A \supseteq \mathcal{S}_{cut_\psi(A)}$.*

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. □

Corollary 6.5 (Thresholds and $\mathcal{S}_A^{dk}$ (1)). *For each ψ, we have $\mathcal{S}_A^{dk} \supseteq \mathcal{S}_{cut_\psi(A)}^{dk}$.*

Proof. It follows from the definition of $\mathcal{S}_A^{dk}$ and from Proposition 6.4. □

Theorem 6.6 (Thresholds and $\mathcal{S}_A$ (2)). *Let $\psi \sqsubseteq glb\{\sigma \in \mathcal{S}_A\}$. Then $\mathcal{S}_A = \mathcal{S}_{cut_\psi(A)}$.*

Proof. By Proposition 6.4, we have $\mathcal{S}_A \supseteq \mathcal{S}_{cut_\psi(A)}$. Moreover, since ψ is lower than all σ in $\mathcal{S}_A$, by Theorem 6.3 we have that all the computations are also in $\mathcal{S}_{cut_\psi(A)}$. So, the statement follows. □

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

Theorem 6.7. *Let $\sigma \in \mathcal{S}_A$ and $\sigma \notin \mathcal{S}_{cut_\psi(A)}$. Then we have $\sigma \sqsubset \psi$.*

Proof. If $\sigma \in \mathcal{S}_A$ and $\sigma \notin \mathcal{S}_{cut_\psi(A)}$, 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. □

The following theorem relates thresholds and $\mathcal{S}_A^{dk}$.

Theorem 6.8 (Thresholds and $\mathcal{S}_A^{dk}$ (2)). *Let $\Psi_A = \{\sigma \in \mathcal{S}_A \mid \nexists \sigma' \in \mathcal{S}_A$ with $\sigma' \sqsupset \sigma\}$ (that is, Ψ_A is the set of "greatest" elements of $\mathcal{S}_A$). Let also $\psi \sqsubseteq glb\{\sigma \in \Psi_A\}$. Then $\mathcal{S}_A^{dk} = \mathcal{S}_{cut_\psi(A)}^{dk}$.*

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 $\psi \sqsubseteq glb\{\sigma \in \Psi_A\}$ we do not eliminate any computation. So we obtain $\mathcal{S}_A^{dk} = \bigoplus_{\sigma \in \mathcal{S}_A} \sigma = \bigoplus_{\sigma \in \Psi_A} \sigma = \mathcal{S}_{cut_\psi(A)}^{dk}$. □

Lemma 6.9. *Given any constraint ψ, we have:*

$$\mathcal{S}_A^{dk} \sqsubseteq \psi \oplus \mathcal{S}_{cut_\psi(A)}^{dk}.$$

Proof. $\mathcal{S}_A^{dk} = lub(\mathcal{S}_A)$ and $\mathcal{S}_{cut_\psi(A)}^{dk} = lub(\mathcal{S}_{cut_\psi(A)})$. Easily to see, $\mathcal{S}_{cut_\psi(A)} \subseteq \mathcal{S}_A$. The solutions that have been eliminated by the cut ψ (that is, all the $\sigma \in \mathcal{S}_A - \mathcal{S}_{cut_\psi(A)}$) are all lower than ψ by Theorem 6.7. So, it easily follows that $\mathcal{S}_A^{dk} \sqsubseteq \psi \oplus \mathcal{S}_{cut_\psi(A)}^{dk}$. □

Theorem 6.10. *Given any constraint ψ, we have:*

$$\mathcal{S}_{cut_\psi(A)}^{dk} \sqsubseteq \mathcal{S}_A^{dk} \sqsubseteq \psi \oplus \mathcal{S}_{cut_\psi(A)}^{dk}.$$

Proof. From Corollary 6.5, we have $\mathcal{S}_{cut_\psi(A)}^{dk} \sqsubseteq \mathcal{S}_A^{dk}$. From Lemma 6.9 we have instead $\mathcal{S}_A^{dk} \sqsubseteq \psi \oplus \mathcal{S}_{cut_\psi(A)}^{dk}$. □

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 $\langle success, \sigma_i \rangle$, 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 σ_1. Then we restart the execution applying the cut level σ_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 $\sigma_1, \ldots, \sigma_k$, we restart execution with cut level $\psi = \sigma_1 \oplus \ldots \oplus \sigma_k$. 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 $\mathcal{S}_{cut_\psi(A)}^{dk} = \psi$ we necessarily have $\mathcal{S}_{cut_\psi(A)}^{dk} = \mathcal{S}_A^{dk} = \psi$).

In a way, such an execution method, that we will call algorithm $A1$, resembles a branch & bound strategy, where the cut levels have the role of the bounds. Notice also that since classical crisp constraints can be represented in the soft CSP framework using a suitable semiring, all the branch & bound results could easily be extended also to the original *cc*.

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:*

$$\mathcal{S}_A^{dk} = \psi \oplus \mathcal{S}_{cut_\psi(A)}^{dk}.$$

PROOF. It easily comes from Theorem 6.10. □

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

First, we start by considering a similar algorithm, that we call $A2$. At the first step, it executes agent A and computes σ_1. At step $i+1$, it executes agent $cut_{\sigma_i}(A)$, where σ_i is the solution computed at step i. When no more solutions are found, the algorithm terminates by collecting the set of all solutions found in the set Ψ'_A.

We will now prove that this algorithm is correct, that is, $\bigoplus_{\sigma \in \Psi'_A} \sigma = \mathcal{S}_A^{dk}$.

If the algorithm stops after one step, what we have to prove is $\mathcal{S}_A^{dk} = \sigma_1 \oplus \mathcal{S}_{cut_{\sigma_1}(A)}^{dk}$. Since σ_1 is for sure lower than $\mathcal{S}_A^{dk}$, this is true by Corollary 6.11.

Let us now assume that the algorithm stops after k steps and collects a set of solutions $\Psi'_A = \{\sigma_1, \ldots, \sigma_k\}$. We recall that $\mathcal{S}_A^{dk} = \bigoplus_{\sigma \in \Psi_A} \sigma$ where $\Psi_A = \{\sigma \in \mathcal{S}_A \mid \nexists \sigma' \in \mathcal{S}_A \text{ with } \sigma' \sqsupset \sigma\}$. Easily to see, Ψ'_A is a superset of Ψ_A, since we could collect a final state σ_i before computing a final state $\sigma_j \sqsupseteq \sigma_i$; in this case both will be present in Ψ'_A but only σ_j will be present in Ψ_A. However, even if Ψ'_A contains more elements than Ψ_A, we have $\bigoplus_{\sigma \in \Psi'_A} = \bigoplus_{\sigma \in \Psi_A}$, since $+$ is idempotent and it is the least upper bound of the lattice.

We now notice that the only difference between algorithms $A2$ and $A1$ is that, at each step, $A1$ performs a cut by using the sum of all the previously computed final states. This means that, at each step, $A1$ can eliminate more computations, but, by the results of Theorem 6.7, the eliminated computations do not change the final result. So, since we proved $A2$ to be correct, $A1$ is correct too.

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 transition rules of Table IV to those of Table III.

(Valued)tell$_1$/ask$_1$. The failing rule for ask and tell simply checks if the added/checked constraint c is *inconsistent* with the store σ 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 ϕ). 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.

Nondeterminism$_1$. The computation fails only when all the branches fail.

Parallelism$_1$. 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.

Table IV. Failure in the scc language

$\dfrac{\sigma \otimes c \sqsubset \phi}{\langle tell(c) \rightarrow_{\phi} A, \sigma\rangle \longrightarrow fail}$	(Tell$_1$)
$\dfrac{(\sigma \otimes c) \Downarrow_{\emptyset} < a}{\langle tell(c) \rightarrow^{a} A, \sigma\rangle \longrightarrow fail}$	(Valued-tell$_1$)
$\dfrac{\sigma \sqsubset \phi}{\langle ask(c) \rightarrow_{\phi} A, \sigma\rangle \longrightarrow fail}$	(Ask$_1$)
$\dfrac{\sigma \Downarrow_{\emptyset} < a}{\langle ask(c) \rightarrow^{a} A, \sigma\rangle \longrightarrow fail}$	(Valued-ask$_1$)
$\dfrac{\langle E_1, \sigma\rangle \longrightarrow fail}{\begin{array}{c}\langle E_1 + E_2, \sigma\rangle \longrightarrow \langle E_2, \sigma\rangle \\ \langle E_2 + E_1, \sigma\rangle \longrightarrow \langle E_2, \sigma\rangle\end{array}}$	(Nondeterminism$_1$)
$\dfrac{\langle A_1, \sigma\rangle \longrightarrow fail}{\begin{array}{c}\langle A_1 \Vert A_2, \sigma\rangle \longrightarrow fail \\ \langle A_2 \Vert A_1, \sigma\rangle \longrightarrow fail\end{array}}$	(Parallelism$_1$)

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]:

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

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 semantics of this agent is $fail$. In fact, the transition rules we have defined do not consider *hang* and infinite computations. Similar semi-decibility results for soft constraint logic programming are proven in [Bistarelli et al. 2001].

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 set of possible moves a process can perform is represented by a constraint. The constraint assigns to each possible move a semiring element representing the cost of that particular move.

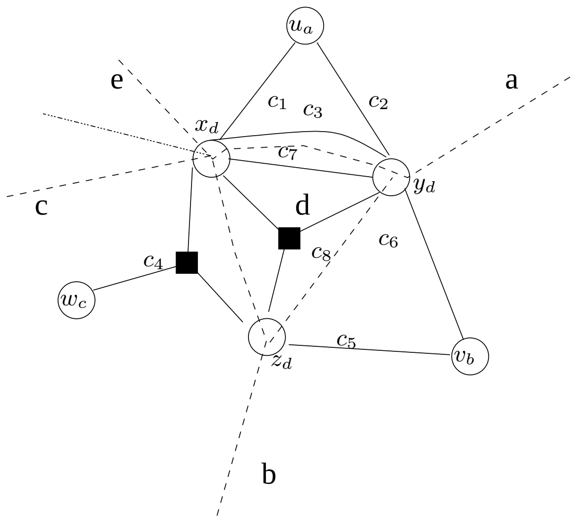


Fig. 2. The SCSP describing part of a process network

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) all processes try to perform a non-idle move.

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.

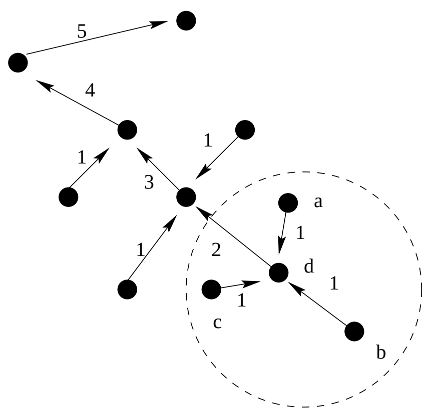
This semiring allows us to model the optimization criterion described above. However, it is also possible to add other optimization criteria, via the pairing of the above semiring with a different one (see section 2.2.2. For example if want to also assign a level of importance to each move, and to maximize the minimum level of importance, we can pair the above semiring with the fuzzy one.

Figure 2 gives the details of a part of a network and it shows eight processes (that is, $c_1, \ldots, c_8$) 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.

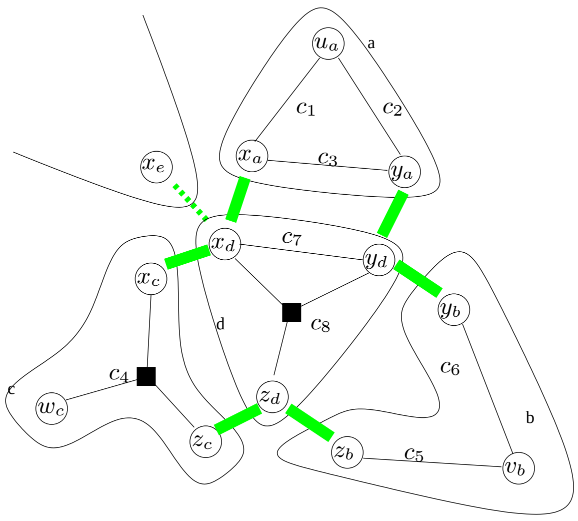
As desired, finding the best solution for the SCSP representing the current state of the process network means finding a move for all the processes such that they perform the same action on the shared variables, the overall cost of the moves is minimized, and there is no idle process. However, since 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. This avoids backtracking. In fact, in a tree-like net structure, as the one of this example, there is no need to backtrack: if pieces are solved bottom-up, there is no failure. Therefore, this solution scheme is an instance of dynamic programming where the small subproblems consist of the constraints in the single sites. Solving the subproblems and combining their solution (using agent A_d) is enough to obtain the global optimal solution.

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. It is important to highlight that we do not need to perform any backtracking steps for the synchronization of the several local computations. The computation is performed independently in each locations. Only later the resulting constraint stores are com-



(a) A possible tree structure for our network.



(b) The SCSP partitioned over the four locations.

Fig. 3. The ordered process network

bined giving raise to the final store. The final store represent the combination of all the requirement imposed in a distributive fashion over the locations.

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

$$A_a : \exists_{u_a}(tell(c_1(x_a,u_a) \wedge c_2(u_a,y_a) \wedge c_3(x_a,y_a)) \rightarrow_{\phi_1} tell(end_a = true) \rightarrow success)$$
$$A_b : \exists_{v_b}(tell(c_5(y_b,v_b) \wedge c_6(z_b,v_b)) \rightarrow_{\phi_2} tell(end_b = true) \rightarrow success)$$
$$A_c : \exists_{w_c}(tell(c_4(x_c,w_c,z_c)) \rightarrow_{\phi_3} tell(end_c = true) \rightarrow success)$$
$$A_d : ask(end_a = true \wedge end_b = true \wedge end_c = true \wedge) \rightarrow_{\phi}$$
$$tell(c_7(x_d,y_d) \wedge c_8(x_d,y_d,z_d) \wedge x_a = x_d = x_c \wedge y_a = y_d = y_b \wedge z_b = z_d = z_c)$$
$$\rightarrow success)$$
$$A : A_a \mid A_b \mid A_c \mid A_d$$

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 a level of 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).

The thresholds ϕ_i of the first three agents are used to stop the computation locally if the best way to assign values to the local variables is not good enough (at least ϕ_i). In this way, the synchronization among sites is not perfomed if a local agent discovers that there is no satisfactory scenario. If all local agents pass their threshold check, then the last agent (A_d) can perform the syncronization. However, it can use a threshold as well (ϕ) to avoid the syncronization because of its own satisfactory notion. Notice that the four agents can have different thresholds, since they run on different sites with possibly different policies. For example, different administrative domains over the Internet (which would be represented by sites a,b,c,d in this example) can have different regulations over quality of services.

8. CONCLUSIONS AND FUTURE WORK

We started our work by realizing the need for handling preferences in Web-related scenarios. To address this need, we have defined soft cc, where soft constraints can be used both at the solver level, to make the notion of consistency more tolerant, and at the language level, to provide an explicit way to deal with approximations and satisfaction levels.

We see soft cc as a first step towards the possibility of using high level declarative languages for Web programming. Of course there are many more aspects to consider to make the language rich enough to be practically usable. However, soft constraints have already shown their usefulness in describing security protocols (see [Bella and Bistarelli 2001; 2002]) and integrity policies (see [Bistarelli and Foley 2003a; 2003b]). We are already considering the introdution of some other features in the language. For instance we could extend the Semantics of the language by observing only the semiring levels associated to the blevel of the final stores. We could observe all the reachable semiring levels (in a fashion similar to the set $\mathcal{S}_A$), or only the best one (in a fashion similar to the set $\mathcal{S}_A^{dk}$). We plan also to investigate the possibility to use techniques similar to those described in section 6.1 to cut computations which are useless w.r.t. this new observables.

We are also considering the possibility of adding soft cc primitives inside other concurrent frameworks, such as Klaim [Nicola et al. 1998] or KAOS [De Nicola et al. 2003] where soft constraints, combined with name unification, are already used for defining access rights and costs in routing.

ACKNOWLEDGMENTS

We are indebted to Paolo Baldan and to Alessandra Di Pierro for valuable suggestions. We also thank the anonymous referees for many useful comments that allowed us to improve the paper.

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.

Bella, G. and Bistarelli, S. 2001. Soft constraints for security protocol analysis: Confidentiality. In *Proc. 3nd International Symposium on Practical Aspects of Declarative Languages (PADL'01)*. Lecture Notes in Computer Science (LNCS), vol. 1990. Springer, 108–122.

Bella, G. and Bistarelli, S. 2002. Confidentiality levels and deliberate/indeliberate protocol attacks. In *Proc. 10th Cambridge International Security Protocol Workshop (CISPW2002)*. Lecture Notes in Computer Science (LNCS), vol. 2845. Springer, 104–119.

Bistarelli, S. 2001. Soft constraint solving and programming: a general framework. Ph.D. thesis, Dipartimento di Informatica, Università di Pisa, Italy.

Bistarelli, S. and Foley, S. 2003a. Analysis of integrity policies using soft constraints. In *Proc. IEEE 4th international Workshop on Policies for Distributed Systems and Networks (POLICY2003)*. IEEE, Lake Como, Italy, 77–80.

Bistarelli, S. and Foley, S. 2003b. A constraint framework for the qualitative analysis of dependability goals: Integrity. In *Proc. 22th international Conference on Computer safety, Reliability and Security (SAFECOMP2003)*. Lecture Notes in Computer Science (LNCS), vol. 2788. Springer, 130–143.

Bistarelli, S., Montanari, U., and Rossi, F. 1995. Constraint Solving over Semirings. In *Proc. 14th International Joint Conference on Artificial Intelligence (IJCAI95)*. Morgan Kaufman, San Francisco, CA, USA, 624–630.

Bistarelli, S., Montanari, U., and Rossi, F. 1997. Semiring-based Constraint Solving and Optimization. *Journal of the ACM (JACM) 44,* 2, 201–236.

Bistarelli, S., Montanari, U., and Rossi, F. 2001. Semiring-based Constraint Logic Programming: Syntax and Semantics. *ACM Transactions on Programming Languages and System (TOPLAS) 23,* 1, 1–29.

Boer, F. D. and Palamidessi, C. 1991. A fully abstract model for concurrent constraint programming. In *Proc. Colloquium on Trees in Algebra and Programming (CAAP91)*. Lecture Notes in Computer Science (LNCS), vol. 493. Springer, 296–319.

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 *Proc. 7th IEEE International Conference on Computer, Communications and Networks (ICCCN'98)*. 614–621.

Clark, D. 1989. Rfc1102: Policy routing in internet protocols. Tech. rep., Network Working Group.

de Boer, F. and Palamidessi, C. 1994. From Concurrent Logic Programming to Concurrent Constraint Programming. In *Advances in Logic Programming Theory*. Oxford University Press, 55–113.

De Nicola, R., Ferrari, G., Montanari, U., Pugliese, R., and Tuosto, E. 2003. A formal basis for reasoning on programmable QoS. In *Verification–Theory and Practice*, N. Dershowitz, Ed. Lecture Notes in Computer Science (LNCS), vol. 2772. Springer, 436–479.

Dubois, D., Fargier, H., and Prade, H. 1993. The calculus of fuzzy restrictions as a basis for flexible constraint satisfaction. In *Proc. IEEE International Conference on Fuzzy Systems*. IEEE, 1131–1136.

Fargier, H. and Lang, J. 1993. Uncertainty in constraint satisfaction problems: a probabilistic approach. In *Proc. European Conference on Symbolic and Qualitative Approaches to Reasoning and Uncertainty (ECSQARU)*. Lecture Notes in Computer Science (LNCS), vol. 747. Springer, 97–104.

Freuder, E. and Wallace, R. 1992. Partial constraint satisfaction. *Artificial Intelligence 58*, 21–70.

Jain, R. and Sun, W. 2000. QoS/Policy/ConstraintBased routing. In *Carrier IP Telephony 2000 Comprehensive Report*. International Engineering Consortium.

Nicola, R. D., Ferrari, G. L., and Pugliese, R. 1998. Klaim: a kernel language for agents interaction and mobility. *IEEE Transactions on Software Engineering (Special Issue on Mobility and Network Aware Computing) 24,* 5, 315–330.

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 *Proc. 3rd IEEE International Conference on Fuzzy Systems*. 1263–1268.

Saraswat, V. 1993. *Concurrent Constraint Programming*. MIT Press.

Saraswat, V., Rinard, M., and Panangaden, P. 1991. Semantic foundations of concurrent constraint programming. In *Proc. 18th ACM Symposium on Principles of Programming Languages (POPL91)*. ACM, 333–352.

Schiex, T. 1992. Possibilistic constraint satisfaction problems, or "how to handle soft constraints?". In *Proc. 8th Conference of Uncertainty in Artificial Intelligence (UAI92)*. Morgan Kaufmann, 269–275.

Schiex, T., Fargier, H., and Verfaille, G. 1995. Valued Constraint Satisfaction Problems: Hard and Easy Problems. In *Proc. 14th International Joint Conference on Artificial Intelligence (IJCAI95)*. Morgan Kaufmann, San Francisco, CA, USA, 631–637.

Scott, D. 1982. Domains for denotational semantics. In *Proc. 9th International Colloquium on Automata, Languages and Programming (ICALP82)*. Vol. 140. Springer, 577–613.

Received August 2002; revised October 2003 and March 2004; accepted June 2004