Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science

Electronic Notes in Theoretical Computer Science 236 (2009) 147–162

www.elsevier.com/locate/entcs

A Nonmonotonic Soft Concurrent Constraint Language for SLA Negotiation

Stefano Bistarelli[a,c,1] Francesco Santini[b,c,2]

[a] Dipartimento di Scienze, Università "G. d'Annunzio" di Chieti-Pescara, Pescara, Italy

[b] IMT Lucca - School for Advanced Studies, Lucca, Italy

[c] Istituto di Informatica e Telematica - CNR, Pisa, Italy

Abstract

We present an extension of the *Soft Concurrent Constraint* language that allows the nonmonotonic evolution of the constraint store. To accomplish this, we introduce some new operations: the $retract(c)$ reduces the current store by c, the $update_X(c)$ transactionally relaxes all the constraints of the store that deal with the variables in the set X, and then adds a constraint c; the $nask(c)$ tests if c is not entailed by the store. We present this framework as a possible solution to the management of resources (e.g. web services and network resource allocation) that need a given *Quality of Service* (*QoS*). The QoS requirements of all the parties should converge, through a negotiation process, on a formal agreement defined as the *Service Level Agreement*, which specifies the contract that must be enforced. c-semirings are the algebraic structures that we use to model QoS metrics.

Keywords: soft constraint logic programming, nonmonotonicity, quality of service, service level agreement

1 Motivations

Many real-life problems require computation mechanisms which are nonmonotonic in their nature. Consider for example an everyday scenario where clients need to reserve some resources, and service providers must allocate those resources providing also a desired *Quality of Service* (QoS). Negotiation [14] is the process by which a group of agents communicate among themselves and try to come to a mutually acceptable agreement on some matter. The means for achieving this goal consist in offering concessions and retracting proposals. When agents are autonomous and cooperation/coordination is attempted at run-time, automated negotiation represents a complex process [14]. Notice that this process must be dynamic because clients and providers can change their requirements during their execution.

[1] Email: bista@sci.unich.it and stefano.bistarelli@iit.cnr.it

[2] Email: f.santini@imtlucca.it and francesco.santini@iit.cnr.it

1571-0661/© 2009 Elsevier B.V. Open access under CC BY-NC-ND license.

doi:10.1016/j.entcs.2009.03.020

To model and manage automated negotiation, in this paper we propose the *Nonmonotonic Soft Concurrent Constraint* (*nmsccp*) language, which extends *Soft Concurrent Constraint Programming* (*sccp*) [3,7] in order to support the nonmonotonic evolution of the constraint store. In classical *sccp* the *tell* and *ask* agents can be equipped with a preference (or consistency) threshold which is used to determine their success, failure, or suspension: the action is enabled only if the store is "consistent enough" with respect to the threshold. Since constraints can only be accumulated (via the *tell* operation), this consistency level can only monotonically decrease starting from the initial empty store: the function used to combine the constraints, i.e. the $\times$ of the semiring, is intensive [6]. To go further, we propose some new actions that provide the user with explicit nonmonotonic operations which can be used to retract constraints from the store (i.e. *update* and *retract*), and a particular *ask* operation (i.e. *nask*), enabled only if the current store does not entail a given constraint.

The *nmsccp* language has two main difference with regard to the classical *sccp*: *i)* the consistency level of the store can be increased by retracting constraints (i.e. it is not monotonic), and *ii)* some of the failures are transformed in suspension because of the nonmonotonicity of the store. According to *i)*, we have extended the semantics of the actions to include also an upper bound on the store consistency (since it can be increased by a *retract*, for example), in order to prune also "too good" computations obtained at a given step. In this way, now we are able to model intervals of acceptability, while in *sccp* there is only a check on "not good enough" computations, i.e. decreasing too much the consistency w.r.t the lower threshold. This leads to *ii)*: in *sccp* an agent fails if the resulting store is not consistent enough with respect to the threshold (i.e. a given semiring value or soft constraint); in *nmsccp* the same agent simply suspends waiting for a possible consistency increase of the current store, which enables the pending action.

We apply these extensions to model *Service Level Agreements* (*SLA*s) [2,15] and their negotiation: soft constraints represent the needs of the agents on the traded resources and the consistency value of the store represents a feedback on the current agreement. In other words, how much all the requirements are consistent among themselves, or how much the global satisfaction is being met. The thresholds on the actions are used to check this interval of preference values, and having a feedback value which is not a plain "yes or no" (i.e. true or false, as in crisp constraints) is clearly more informative. Using soft constraints (e.g. "at most *around* 10 Mbyte of bandwidth") gives the service provider and clients more flexibility in expressing their requests with respect to crisp constraints (e.g. "*exactly* 10 Mbyte"), and therefore there are more chances to reach a shared agreement. Moreover, the cost model is very adaptable to the specific problem, since it is parametric with the chosen semiring, and its semantics is directly embedded in the requirement definition itself (i.e. the constraint) and in the language modeling the agent (e.g. the thresholds on the *tell* and *retract* actions).

The remainder of this paper is organized as follows. In Sec. 2 we summarize the background information. Sec. 3 features the nonmonotonic language, its operational

semantics and how the consistency intervals are managed. In Sec. 4 we show how the language can be used to represent preference-driven negotiations. At last, Sec. 5 shows related works and Sec. 6 concludes by indicating future research directions.

Related Work on Nonmonotonic Extensions.

The inspiration for this work comes from [9] and [11]: in [11] the authors present a nonmonotonic framework for *Concurrent Constraint Programming* (*ccp*) [20], together with its semantics. Our *nask* and *update* operations (see Sec. 3) are the soft versions of those described in [11], while the *atell*, which adds a constraint only if it is consistent with the store, can be trivially modelled with the classical (valued) *tell* of *sccp*. A negative ask like our *nask* is described also in [19]. The idea for a fine-grained removal of constraints (the *retract* in Sec. 3) comes from [9], which describes a different nonmonotonic framework for *ccp*. Its main purpose was not to add any additional nondeterminism (besides the choice operator) by keeping track of the dependencies among constraints in the same parallel computation, otherwise the nonmonotonic evolution could yield different results if executed with different scheduling policies. However, in our language we decided to allow this kind of nondeterminism, since we believe it is more natural to experience this behaviour during the negotiation interactions in open systems. Other examples of nonmonotonic evolution of the constraint store in *ccp* are presented in [13], and their line of research is usually called *Linear Concurrent Constraint Programming*.

2 Background

Absorptive Semiring.

An absorptive semiring [5] S can be represented as a $\langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$ tuple such that: *i)* A is a set and $\mathbf{0}, \mathbf{1} \in A$; *ii)* $+$ is commutative, associative and $\mathbf{0}$ is its unit element; *iii)* $\times$ is associative, distributes over $+$, $\mathbf{1}$ is its unit element and $\mathbf{0}$ is its absorbing element. Moreover, $+$ 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 [6]): *i)* $\leq_S$ is a partial order; *ii)* $+$ and $\times$ are monotonic on $\leq_S$; *iii)* $\mathbf{0}$ is its minimum and $\mathbf{1}$ its maximum; *iv)* $\langle A, \leq_S \rangle$ is a complete lattice and, for all $a, b \in A$, $a + b = lub(a, b)$ (where lub is the *least upper bound*). Informally, the relation $\leq_S$ gives us a way to compare semiring values and constraints. In fact, when we have $a \leq_S b$ (or simply $a \leq b$ when the semiring will be clear from the context), we will say that b *is better than* a.

In [5] the authors extended the semiring structure by adding the notion of *division*, i.e. $\div$, as a weak inverse operation of $\times$. An absorptive semiring S is *invertible* if, for all the elements $a, b \in A$ such that $a \leq b$, there exists an element $c \in A$ such that $b \times c = a$ [5]. If S is absorptive and invertible, then, S is *invertible by residuation* if the set $\{x \in A \mid b \times x = a\}$ admits a maximum for all elements $a, b \in A$ such that $a \leq b$ [5]. Moreover, if S is absorptive, then it is *residuated* if the set $\{x \in A \mid b \times x \leq a\}$ admits a maximum for all elements $a, b \in A$, denoted $a \div b$. With an abuse of notation, the maximal element among solutions is denoted $a \div b$.

This choice is not ambiguous: if an absorptive semiring is invertible and residuated, then it is also invertible by residuation, and the two definitions yield the same value.

To use these properties, in [5] it is stated that if we have an absorptive and complete semiring [3], then it is residuated. For this reason, since all classical soft constraint instances (i.e. *Classical CSPs*, *Fuzzy CSPs*, *Probabilistic CSPs* and *Weighted CSPs*) are complete and consequently residuated, the notion of semiring division can be applied to all of them. Therefore, for all these semirings it is possible to use the $\div$ operation as a "particular" inverse of $\times$; its extension to soft constraints, defined as $\oplus\!\!\!\!-$, can be used to (partially) remove soft constraints from the store (see next Paragraph).

Soft Constraint System.

A *soft constraint* [6,3] may be seen as a constraint where each instantiation of its variables has an associated preference. Given $S = \langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$ and an ordered set of variables V over a finite domain D, a soft constraint is a function which, given an assignment $\eta : V \rightarrow D$ of the variables, returns a value of the semiring. Using this notation $\mathcal{C} = \eta \rightarrow A$ is the set of all possible constraints that can be built starting from S, D and V.

Any function in $\mathcal{C}$ involves all the variables in V, but we impose that it depends on the assignment of only a finite subset of them. So, for instance, a binary constraint $c_{x,y}$ over variables x and y, is a function $c_{x,y} : (V \rightarrow D) \rightarrow A$, but it depends only on the assignment of variables $\{x, y\} \subseteq V$ (the *support* of the constraint, or *scope*). Note that $c\eta[v := d_1]$ means $c\eta'$ where η' is η modified with the assignment $v := d_1$. Notice also that, with $c\eta$, the result we obtain is a semiring value, i.e. $c\eta = a$.

Given the set $\mathcal{C}$, the combination function $\otimes : \mathcal{C} \times \mathcal{C} \rightarrow \mathcal{C}$ is defined as $(c_1 \otimes c_2)\eta = c_1\eta \times c_2\eta$ (see also [6,3,7]). Having defined the operation $\div$ on semirings, the constraint division function $\oplus\!\!\!\!- : \mathcal{C} \times \mathcal{C} \rightarrow \mathcal{C}$ is instead defined as $(c_1 \oplus\!\!\!\!- c_2)\eta = c_1\eta \div c_2\eta$ [5]. Informally, performing the $\otimes$ or the $\oplus\!\!\!\!-$ between two constraints means building a new constraint whose support involves all the variables of the original ones, and which associates with each tuple of domain values for such variables a semiring element which is obtained by multiplying or, respectively, dividing the elements associated by the original constraints to the appropriate sub-tuples. The partial order $\leq_S$ over $\mathcal{C}$ can be easily extended among constraints by defining $c_1 \sqsubseteq c_2 \iff c_1\eta \leq c_2\eta$. Consider the set $\mathcal{C}$ and the partial order $\sqsubseteq$. Then an entailment relation $\vdash \subseteq \wp(\mathcal{C}) \times \mathcal{C}$ is defined s.t. for each $C \in \wp(\mathcal{C})$ and $c \in \mathcal{C}$, we have $C \vdash c \iff \bigotimes C \sqsubseteq c$ (see also [3,7]).

Given a constraint $c \in \mathcal{C}$ and a variable $v \in V$, the *projection* [6,3,7] of c over $V \backslash \{v\}$, written $c \Downarrow_{(V \backslash \{v\})}$ is the constraint c' s.t. $c'\eta = \sum_{d \in D} c\eta[v := d]$. Informally, projecting means eliminating some variables from the support. This is done by associating with 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

[3] If S is an absorptive semiring, then S is complete if it is closed with respect to infinite sums, and the distributivity law holds also for an infinite number of summands.

of this tuple over the eliminated variables. To treat the hiding operator of the language, a general notion of existential quantifier is introduced by using notions similar to those used in cylindric algebras. For each $x \in V$, the hiding function [3,7] is defined as $(\exists_x c)\eta = \sum_{d_i \in D} c\eta[x := d_i]$.

To model parameter passing, for each $x, y \in V$ a diagonal constraint [3,7] is defined as $d_{xy} \in \mathcal{C}$ 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 $a \neq b$. Considering 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 $\mathcal{C}$, then $S_C = \langle \mathcal{C}, \otimes, \bar{\mathbf{0}}, \bar{\mathbf{1}}, \exists_x, d_{xy} \rangle$ [4] is a cylindric constraint system (*"a la Saraswat"* [5] [7]).

3 The Language

In this Section we will give the flavour of the new operations and the reasons why we introduced them in this new language. Then we will show the entire language together with its operational semantics and some simple examples to illustrate the evolution of the agent computation.

The $retract(c)$ operation is at the basis of our nonmonotonic extension of the *sccp* language, since it permits to remove the constraint c from the current store σ. It is worth to notice that our *retract* can be considered as a "relaxation" of the store, and not only as a strict removal of the token representing the constraint, because in soft constraints we do not have the concept of token. Thus if c (parameter of retract) satisfies $\sigma \sqsubseteq c$ then it can be removed, even if c is different from any other constraints previously added to σ.

To use a metaphor describing the sequence of actions, imagine to pour a liquid into and out a bowl with a spoon. The content of the bowl represents the store, and the liquid in the spoon represents the soft constraint we want to add and retract from the store; as the two liquids are mixed, we lose the identity of the added soft constraint, which can worsen the condition of the store by raising the level of the liquid in the bowl. When we want to relax the store, we remove some of the liquid with the spoon, and that corresponds to the removed constraint: the consistency is incremented because the level of the bowl is lowered. This "bowl example" is appropriate when $\times$ is not idempotent, otherwise pouring the same constraint multiple times would not increase the liquid level.

The $update_X(c)$ primitive has been inspired by the work in [11]. It consists in a sort of "assignment" operation, since it transactionally relaxes all the constraints of the store that deal with variables in the set X, and then adds a constraint c (usually with $support = X$). This operation is variable-grained with respect to our *retract*, and for many applications (as ours, on SLA negotiation), it is very convenient to have a relaxation operation that is focused on one (or some) variable: the reason is that it could be required to completely renew the knowledge about a parameter (e.g. the bandwidth of the example in Sec. 4).

[4] $\bar{\mathbf{0}}$ and $\bar{\mathbf{1}}$ respectively represent the constraints associating $\mathbf{0}$ and $\mathbf{1}$ to all assignments of domain values; in general, the $\bar{a}$ function returns the semiring value a.

[5] Notice that in *sccp*, algebraicity is not required, since the algebraic nature of $\mathcal{C}$ strictly depends on the properties of the semiring [7].

$$P ::= \quad F.A$$

$$F ::= \quad p(Y) :: A \mid F.F$$

$$A ::= \quad success \mid tell(c) \rightarrowtail A \mid retract(c) \rightarrowtail A \mid update_X(c) \rightarrowtail A \mid E \mid A \| A \mid \exists x.A \mid p(Y)$$

$$E ::= \quad ask(c) \rightarrowtail A \mid nask(c) \rightarrowtail A \mid E + E$$

Fig. 1. Syntax of the nmsccp language.

The $nask(c)$ operation (crisp examples are in [9,16]) is enabled only if the current store does not entail c; it is the negative version of *ask*, since it detects *absence* of information. Note that, in general, $ask(\neg c)$ is different from $nask(c)$, so it is necessary to introduce a completely new primitive. Consider for example the store $\{x \leq 10\}$: while the action $nask(x < 5)$ succeeds, $ask(x \geq 5)$ would block the computation. Consider also that the notion of $\neg c$ (i.e. the negation of a constraint) is not always meaningful with preferences based on semirings, except, for instance, for the *Boolean* semiring (i.e. $\langle \{0, 1\}, \vee, \wedge, 0, 1\rangle$). It would be difficult to define $\neg c$ when using *Weighted* semirings [3,6]. This operation improves the expressivity of the language, since it allows to check facts not yet derivable from the store (it can be valuable to add them), or no longer derivable (to check if some constraints have been removed), or facts that we do not want to be implied by the store.

Given a soft constraint system as defined in Sec. 2 and any related constraint c, the syntax of agents in *nmsccp* is given in Fig. 1. 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, X is a set of variables and Y is a tuple of variables.

In addition to the new operations, the other most important variation with regard to *sccp* is the action prefixing symbol $\rightarrowtail$ in the syntax notation, which can be considered as a general "checked" transition of the type $\rightarrow^{\varphi_2}_{\varphi_1}$ (e.g., referring to Fig. 1, we can write $ask(c) \rightarrow^{\varphi_2}_{\varphi_1} A$), where φ_i is a placeholder that can stand for either a semiring element a_i or a constraint ϕ_i, i.e. $\varphi_i = a_i/\phi_i$.

In the first case (i.e. a_i), we need to summarize the consistency of the store into a plain value and "compare" it with the a_i semiring value, while in the second case (i.e. ϕ_i), we need to make a pointwise comparison between the store and the ϕ_i constraint, i.e. a comparison between two constraints [7]. The way we compare these values/constraints depends on their level in the transition symbol: a_1 (or ϕ_1) will be used as a cut level to prune computations that at this point are not good enough (i.e. a lower bound), while a_2 (or ϕ_2) to prune computations that are too good (i.e. an upper bound). The four possible instantiation of $\rightarrowtail$ are given in Fig. 2, i.e. $\rightarrow^{a_2}_{a_1}$, $\rightarrow^{\phi_2}_{a_1}$, $\rightarrow^{a_2}_{\phi_1}$ and $\rightarrow^{\phi_2}_{\phi_1}$ (the semantics of these checked transitions will be better explained in Sec. 3.1). As in classical *sccp*, the semiring values a_1 and a_2 represent two *cut levels* that summarize the consistency of the store into a plain value. On the other hand, the constraints ϕ_1 and ϕ_2 represent a finer check of the store, since a pointwise comparison between the store and these constraints is performed.

Therefore, we can now model intervals of acceptability during the computation,

while in classical *sccp* this is not possible: *sccp* being monotonic, since the consistency level of the store can only be decreased during the executions of the agents, it is only meaningful to prune those computations that decrease this level too much. On the other hand, in *nmsccp* there is the possibility to remove constraints from the store, and thus the level can be increased again (this leads to the absence of a fail agent). For this reason we claim the importance of checking also that the consistency level of the store will not exceed a given threshold.

Having an interval of preferences, and not only a lower bound, is very important in negotiation, since it allows to improve the expressivity of requests and results. For instance, consider the preference as a cost for a given resource: the lower threshold of the interval will prevent us from paying that resource too much (i.e. a high cost means a low preference), while the upper threshold models a clause in the contract that forces us to pay at least a minimum price.

The classical *ask* and *tell* operations in *sccp* (where only the lower bound is present) can be obtained also in *nmsccp*: e.g. $ask/tell(c) \rightarrow^{\bar{1}}_{\phi} A$.

3.1 The Operational Semantics

To give an operational semantics to our language we need to describe an appropriate transition system $\langle \Gamma, T, \rightarrow \rangle$, 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 is $\Gamma = \{\langle A, \sigma \rangle\}$, where $\sigma \in \mathcal{C}$ while the set of terminal configurations is instead $T = \{\langle success, \sigma \rangle\}$. The transition rule for the *nmsccp* language are defined in Fig. 3.

The $\rightarrowtail$ is a generic checked transition used by several actions of the language. Therefore, to simplify the rules in Fig. 3 we define a function $check_{\rightarrowtail} : \sigma \rightarrow \{true, false\}$ (where $\sigma \in \mathcal{C}$), that, parametrized with one of the four possible instances of $\rightarrowtail$ (**C1-C4** in Fig. 2), returns *true* if the conditions defined by the specific instance of $\rightarrowtail$ are satisfied, or *false* otherwise. The conditions between parentheses in Fig. 2 claim that the lower threshold of the interval clearly cannot be "better" than the upper one, otherwise the condition is intrinsically wrong.

C1: $\rightarrowtail = \rightarrow^{a_2}_{a_1}$ $\quad check(\sigma)_{\rightarrowtail} = true$ if $\begin{cases} \sigma \Downarrow_{\emptyset} \not>_S a_2 \\ \sigma \Downarrow_{\emptyset} \not<_S a_1 \end{cases}$ (with $a_1 \not> a_2$)

C3: $\rightarrowtail = \rightarrow^{a_2}_{\phi_1}$ $\quad check(\sigma)_{\rightarrowtail} = true$ if $\begin{cases} \sigma \Downarrow_{\emptyset} \not>_S a_2 \\ \sigma \not\sqsubseteq \phi_1 \end{cases}$ (with $\phi_1 \Downarrow_{\emptyset} \not> a_2$)

C2: $\rightarrowtail = \rightarrow^{\phi_2}_{a_1}$ $\quad check(\sigma)_{\rightarrowtail} = true$ if $\begin{cases} \sigma \not\sqsupseteq \phi_2 \\ \sigma \Downarrow_{\emptyset} \not<_S a_1 \end{cases}$ (with $a_1 \not> \phi_2 \Downarrow_{\emptyset}$)

C4: $\rightarrowtail = \rightarrow^{\phi_2}_{\phi_1}$ $\quad check(\sigma)_{\rightarrowtail} = true$ if $\begin{cases} \sigma \not\sqsupseteq \phi_2 \\ \sigma \not\sqsubseteq \phi_1 \end{cases}$ (with $\phi_1 \not\sqsupseteq \phi_2$)

Otherwise, within the same conditions in parentheses, $check(\sigma)_{\rightarrowtail} = false$

Fig. 2. Definition of the *check* function for each of the four checked transitions.

Notice that in Fig. 2 we use $\not<_S a_1$ instead of $\geq_S a_1$ because we can possibly deal with partial orders. Similar considerations can be done for $\not\sqsubset$ instead of $\sqsupseteq$.

Some of the intervals in Fig. 2 (**C1**, **C2** and **C3**) are checked by considering the least upper bound among the values yielded by the solutions of a *Soft Constraint Satisfaction Problem* (SCSP) [3] defined as $P = \langle C, con\rangle$ (C is the set of constraints and $con \subseteq V$, i.e. a subset the problem variables). This is called the *best level of consistency* and it is defined by $blevel(P) = Sol(P) \Downarrow_{\emptyset}$, where $Sol(P) = (\bigotimes C) \Downarrow_{con}$; notice that $supp(blevel(P)) = \emptyset$. We also say that: P is α-consistent if $blevel(P) = \alpha$; P is consistent iff there exists $\alpha >_S \mathbf{0}$ such that P is α-consistent; P is inconsistent if it is not consistent. In Fig. 2 **C1** checks if the α-consistency of the problem is between a_1 and a_2.

In words, **C1** states that we need at least a solution as good as a_1 entailed by the current store, but no solution better than a_2; therefore, we are sure that some solutions satisfy our needs, and none of these solutions is "too good". The semantics of these checks can easily be changed in order to model different requirements on the preference interval, e.g. to guarantee that all the solutions in the store (and not at least one) have a preference contained in the given interval.

R1 $\dfrac{check(\sigma \otimes c)_{\rightarrowtail}}{\langle \mathbf{tell}(c) \rightarrowtail A, \sigma\rangle \longrightarrow \langle A, \sigma \otimes c\rangle}$ **Tell**

R2 $\dfrac{\sigma \vdash c \quad check(\sigma)_{\rightarrowtail}}{\langle \mathbf{ask}(c) \rightarrowtail A, \sigma\rangle \longrightarrow \langle A, \sigma\rangle}$ **Ask**

R3 $\dfrac{\langle A, \sigma\rangle \longrightarrow \langle A', \sigma'\rangle}{\begin{array}{c}\langle A \parallel B, \sigma\rangle \longrightarrow \langle A' \parallel B, \sigma'\rangle \\ \langle B \parallel A, \sigma\rangle \longrightarrow \langle B \parallel A', \sigma'\rangle\end{array}}$ **Parall1**

R4 $\dfrac{\langle A, \sigma\rangle \longrightarrow \langle success, \sigma'\rangle}{\begin{array}{c}\langle A \parallel B, \sigma\rangle \longrightarrow \langle B, \sigma'\rangle \\ \langle B \parallel A, \sigma\rangle \longrightarrow \langle B, \sigma'\rangle\end{array}}$ **Parall2**

R5 $\dfrac{\langle E_j, \sigma\rangle \longrightarrow \langle A_j, \sigma'\rangle \quad j \in [1, n]}{\langle \Sigma_{i=1}^{n} E_i, \sigma\rangle \longrightarrow \langle A_j, \sigma'\rangle}$ **Nondet**

R6 $\dfrac{\sigma \not\vdash c \quad check(\sigma)_{\rightarrowtail}}{\langle nask(c) \rightarrowtail A, \sigma\rangle \longrightarrow \langle A, \sigma\rangle}$ **Nask**

R7 $\dfrac{\sigma \sqsubseteq c \quad \sigma' = \sigma \ominus c \quad check(\sigma')_{\rightarrowtail}}{\langle retract(c) \rightarrowtail A, \sigma\rangle \longrightarrow \langle A, \sigma'\rangle}$ **Retract**

R8 $\dfrac{\sigma' = (\sigma \Downarrow_{(V \setminus X)}) \otimes c \quad check(\sigma')_{\rightarrowtail}}{\langle update_X(c) \rightarrowtail A, \sigma\rangle \longrightarrow \langle A, \sigma'\rangle}$ **Update**

R9 $\dfrac{\langle A[x/y], \sigma\rangle \longrightarrow \langle B, \sigma'\rangle}{\langle \exists x.A, \sigma\rangle \longrightarrow \langle B, \sigma'\rangle}$ with y *fresh* **Hide**

R10 $\dfrac{\langle A, \sigma\rangle \longrightarrow \langle B, \sigma'\rangle}{\langle p(Y), \sigma\rangle \longrightarrow \langle B, \sigma'\rangle}$ $p(Y) :: A \in F$ **P-call**

Fig. 3. The transition system for *nmsccp*.

Here is a description of the transition rules in Fig. 3. In the **Tell** rule (**R1**), if the store $\sigma \otimes c$ satisfies the conditions of the specific $\rightarrowtail$ transition of Fig. 2, then the agent evolves to the new agent A over the store $\sigma \otimes c$. Therefore the constraint c is added to the store σ. The conditions are checked on the (possible) next-step store: i.e. $check(\sigma')_{\rightarrowtail}$.

To apply the **Ask** rule (**R2**), we need to check if the current store σ entails the constraint c and also if the current store is consistent with respect to the lower and upper thresholds defined by the specific $\rightarrowtail$ transition arrow: i.e. if $check(\sigma)_{\rightarrowtail}$ is

true.

Parallelism and nondeterminism: the composition operators $+$ and $\|$ respectively model nondeterminism and parallelism. A parallel agent (rules **R3** and **R4**) will succeed when both agents succeed. This operator is modelled in terms of *interleaving* (as in the classical *ccp*): each time, the agent $A \parallel B$ can execute only one between the initial enabled actions of A and B (**R3**); a parallel agent will succeed if all the composing agents succeed (**R4**). The nondeterministic rule **R5** chooses one of the agents whose guard succeeds, and clearly gives rise to global nondeterminism.

The **Nask** rule is needed to infer the absence of a statement whenever it cannot be derived from the current state: the semantics in **R6** shows that the rule is enabled when the consistency interval satisfies the current store (as for the *ask*), and c is not entailed by the store: i.e. $\sigma \not\sqsubseteq c$.

Retract: with **R7** we are able to "remove" the constraint c from the store σ, using the $\div\!\!\!\!\bigcirc$ constraint division function defined in Sec. 2. According to **R7**, we require that the constraint c is entailed by the store, i.e. $\sigma \sqsubseteq c$. Notice that in [5] the division is instead always defined, but for the *nmsccp* language we decided to be able to remove a quantity c only if the store is "big" enough to permit the removal of c, i.e. we want that $a \div b$ is possible only if $a \leq_S b$. For example, consider the c_1, c_2 and c_3 weighted constraints in Fig. 4: the domain of the variable x is $\mathbb{N}$ and the adopted semiring is instead the classical *Weighted* semiring $\langle \mathbb{R}^+, min, +, +\infty, 0\rangle$. It is possible to perform $c_2 \div\!\!\!\!\bigcirc c_1$ because $c_2 \sqsubseteq c_1$ (the c_1 function is completely dominated by c_2 for every $x \in \mathbb{N}$, and thus c_1 is better), but it is not possible to perform $c_3 \div\!\!\!\!\bigcirc c_1$ because, for $x = 1$ (for instance), $c_3(x) = 2$ is better than $c_1(x) = 4$: thus $2 \leq 4$ and the semiring division $2 \div 4$ cannot consequently be performed because of the **R7** definition. Clearly, it is also possible to completely remove a constraint as if using tokens:

Theorem 3.1 (Complete removal) *Given a soft constraint system $\mathcal{C}$, where the semiring S is invertible by residuation and thus $\div\!\!\!\!\bigcirc$ can be defined, then the* nmsccp *agent $\langle tell(c_i) \rightarrowtail retract(c_i) \rightarrowtail A, \sigma_k\rangle$ is equivalent (i.e. the final store is the same) to $\langle A, \sigma_k\rangle$, for every constraint c_i, store σ_k and $\rightarrowtail$ (if enabled).*

As a sketch of the proof, the agents' equivalence comes from the properties explained in [5], i.e. $a \times b \div b = a$ always holds, given any two elements $a, b \in S$. Since the constraint operations ($\otimes$ and $\div\!\!\!\!\bigcirc$) are derived from their related semiring operators ($\times$ and $\div$), the same properties hold.

The semantics of **Update** rule (**R8**) [11] resembles the assignment operation in imperative programming languages: given an $update_X(c)$, for every $x \in X$ it removes the influence over x of each constraint in which x is involved, and finally a new constraint c is added to the store. To remove the information concerning all $x \in X$, we project (see Sec. 2) the current store on $V \backslash X$, where V is the set of all the variables of the problem and X is a parameter of the rule (projecting means eliminating some variables). If $X = V$, this operation finds the *blevel* of the problem defined by the store, before adding c. At last, the levels of consistency

$$c_1 : (\{x\} \to \mathbb{N}) \to \mathbb{R}^+ \quad \text{s.t. } c_1(x) = x + 3 \qquad c_2 : (\{x\} \to \mathbb{N}) \to \mathbb{R}^+ \quad \text{s.t. } c_2(x) = 2x + 8$$

$$c_3 : (\{x\} \to \mathbb{N}) \to \mathbb{R}^+ \quad \text{s.t. } c_3(x) = 2x$$
$$c_4 : (\{x\} \to \mathbb{N}) \to \mathbb{R}^+ \quad \text{s.t. } c_4(x) = x + 5$$

$$c_5 : (\{x\} \to \mathbb{N}) \to \mathbb{R}^+ \text{ s.t. } c_5(x) = \bar{3} \qquad c_6 : (\{y\} \to \mathbb{N}) \to \mathbb{R}^+ \text{ s.t. } c_6(y) = y + 1$$

Fig. 4. Six weighted soft constraints (notice that $c_2 = c_1 \otimes c_4$).

are checked on the obtained store, i.e. $check(\sigma')_{\rightarrowtail}$. Notice that all the removals and the constraint addition are transactional, since are executed in the same rule. Moreover, notice that the removal semantics of the *update* is quite different from that of the *retract*: the *update* operation can always be applied, while the *retract* can be applied only when $\sigma \sqsubseteq c$. In addition, performing an *update* is different from sequentially performing one (or some) *retract* and then a *tell*: the *retract* relaxes the store in a "clear" way, while the *update* "releases" one (or more) variable x by choosing the best semiring value for each constraint c supported by x (i.e. $\sigma \Downarrow_{(V \setminus \{x\})} = \sum_{d_i \in D} c\eta[x := d_i]$, where D is the domain of x). Therefore, if c is supported also by another variable y, c is somewhat still constraining y after the *update* operation. As an example of the different semantics between an *update* and a *retract*-*tell* sequence, the agent $\langle tell(c_5) \rightarrow^0_\infty retract(c_5) \rightarrow^0_\infty tell(c_2), \bar{0}\rangle$ (in the *Weighted* semiring $\mathbf{1} \equiv \bar{0}$) results in the store $c_5 \div c_5 \otimes c_2 = c_2$, while $\langle tell(c_5) \rightarrow^0_\infty update_{\{x\}}(c_2), \bar{0}\rangle$ results in the store $\bar{3} \otimes c_2$ (i.e. $c_5 \otimes c_2$), where $\bar{3} = c_5 \Downarrow_{(V \setminus \{x\})}$ (see Fig. 4).

Hidden variables: the semantics of the existential quantifier in **R9** is similar to that described in [18] by using the notion of *freshness* of the new variable added to the store.

Procedure calls: the semantics of the procedure call (**R10**) has already been defined in [7]: the notion of diagonal constraints (as defined in Sec. 2) is used to model parameter passing.

Given the transition system proposed in Fig. 3, we define for each agent A the set of final stores that collects the results of successful computations that A can perform (i.e. the *observables*): $\mathcal{S}_A = \{\sigma \Downarrow_{var(A)} | \ \langle A, \bar{\mathbf{1}}\rangle \rightarrow^* \langle success, \sigma\rangle\}$.

No Failures.

The *nmsccp* agents computation can only be successful or can suspend waiting for a change of the store in which it is possible to execute the action on which an agent is suspended on. This represents a further difference with respect to *sccp* where, when trying to execute a (valued or not) *ask*/*tell*, if the resulting level of the store consistency is lower than the threshold labeled on the transition arrow, then this is considered a failure (see [7]): in *sccp* the store consistency can only be monotonically decreased, and therefore a better level can never be reached during the successive steps. In *nmsccp*, another agent in parallel can instead perform a

retract (or an *update*) and can consequently increase the consistency level of the store, then enabling the idle action.

Preference Representation and Operations

The representational and computational issues are complex and would deserve a deep discussion [10]. However, some different considerations can be provided whether or not the language adopted to represent the constraints preference is finite.

As a practical example of (a specific subset of) soft constraints that have a finite representation, consider the *Weighted* semiring and consider a class of constraints whose soft preference (or cost) is represented by a polynomial expression over the variables involved in the constraints. In this case, adding a constraint to the store means to obtain a new polynomial form that is the sum of the new preference and the polynomial representing the current store; retracting a constraint means just to subtract the polynomial form from the store. Suppose we have three constraints $c_1(x, y) = x^2 - 3x + 4y$, $c_2(x) = 3x + 2$ and $c_3(y) = 3y - 2$: if the initial store contains $c_1(x, y)$, $tell(c_2)$ gives $(c_1 \otimes c_2) = x^2 - 3x + 4y + 3x + 2 = x^2 + 4y + 2$, and then a $retract(c_3)$ would result in the store preference $(c_1 \otimes c_2 \ominus c_3) = x^2 + 4y + 2 - (3y - 2) = x^2 + y$. To compute the result of an $update_{\{y\}}(c_4)$ we need to project over $V \setminus \{y\}$ (see Sec. 2) before adding c_4: therefore, if the store preference is $x^2 + y$, we must find the minimum of this polynomial by assigning $y = 0$ and finally obtaining $x^2 \otimes c_4 = x^2 + x + 5$ as result (see Fig. 4). Notice that in the *Weighted* semiring, to maximize the preference means to minimize the polynomial.

Otherwise, if soft constraints have not a finite representation, we can model the store as an ordered list of constraints and actions. For examples, if the agents have chronologically performed the actions $tell(c_1)$, $tell(c_2)$ $retract(c_3)$ and $update_X(c_4)$, the store will be $c_1 \otimes c_2 \ominus c_3 \Downarrow_{(V \setminus X)} \otimes c4$ (whose composition is left-associative). Therefore, at each step it is possible to compute the actual store in order to verify the entailments among constraints and the consistency intervals. Thus, the actions ordering is important:

Theorem 3.2 (Actions ordering) *Given a soft constraint system $\mathcal{C}$, where the semiring S is invertible by residuation, changing the* tell *and* retract *actions ordering inside an agent changes the final store.*

In fact, if we suppose the $\times$ of S as idempotent, we have the *nmsccp* agent $\langle tell(c_i) \rightarrowtail retract(c_i) \rightarrowtail tell(c_i) \rightarrowtail A, \sigma_k \rangle \equiv \langle A, c_i \otimes \sigma_k \rangle$, and by changing the ordering of actions it differs from $\langle tell(c_i) \rightarrowtail tell(c_i) \rightarrowtail retract(c_i) \rightarrowtail A, \sigma_k \rangle \equiv \langle A, \sigma_k \rangle$, for every constraint c_i, store σ_k and $\rightarrowtail$ (if enabled). To prove it, we consider that for every semiring element $a \in S$, we have $(a \times a) \div a = \mathbf{1}$ (since $a \times a = a$, if $\times$ is idempotent), but $(a \div a) \times a = a$. This is due to idempotency of $\times$ and the properties of $\div$ shown in [5]. Theorem 3.2 holds even if $\times$ is not idempotent: for example (see the constraints in Fig. 4), $\langle tell(c_2) \rightarrowtail retract(c_4) \rightarrowtail success, c_1 \rangle$ successfully terminates with the store $c_1 \otimes c_2 \ominus c_4 \equiv 2x + 6$, while $\langle retract(c_4) \rightarrowtail tell(c_2) \rightarrowtail success, c_1 \rangle$ is suspended on the first *retract*, since the $\sigma \sqsubseteq c$ precondition

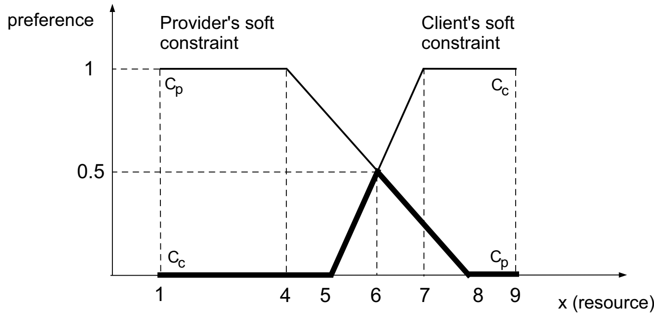

Fig. 5. The graphical interpretation of a fuzzy agreement

of **R7** in Fig. 3 is false (here, $c_1 \sqsubseteq c_4$ is false).

This representation (i.e. keeping also the sequence of operations) differs from the classical one given by Saraswat [18] or in [8], since in these works a *retract* removes from the store only one instance of the token: $\langle tell(c_1) \rightarrow tell(c_1) \rightarrow retract(c_1) \rightarrow A, \bar{\mathbf{1}}\rangle \equiv \langle A, c_1\rangle$, even if $\times$ is idempotent. Therefore, the ordering of the actions is useless and the store can be seen only as a set of tokens.

4 The Negotiation of Service Level Agreements

One of the most meaningful application of the *nmsccp* language is to model generic entities negotiating a formal agreement, i.e. a SLA [2,15]. The main task consists in accomplishing the requests of all the agents by satisfying their QoS needs. Considering the fuzzy negotiation in Fig. 5 (*Fuzzy* semiring: $\langle[0,1], max, min, 0, 1\rangle$) both a provider and a client can add their request to the store σ (respectively $tell(c_p)$ and $tell(c_c)$): the thick line represents the consistency of σ after the composition (i.e. *min*), and the *blevel* of this SCSP (see Sec. 3.1) is the *max*, where both requests intersects (i.e. in 0.5).

We present four short examples to suggest possible negotiation scenarios. We suppose there are two distinct companies (e.g. providers P_1 and P_2) that want to merge their services in a sort of pipeline, in order to offer to their clients a single structured service: e.g. P_1 completes the functionalities of P_2. This example models the *cross-domain* management of services proposed in [2]. The variable x represents the global number of failures they can sustain during the service provision, while the preference models the number of hours (or a money cost in hundreds of euro) needed to manage them and recover from them. The preference interval on transition arrows models the fact that both P_1 and P_2 explicitly want to spend some time to manage the failures (the upper bound in Fig. 2), but no so much time (lower bound in Fig. 2). We will use the *Weighted* semiring and the soft constraints given in Fig. 4. Even if the examples are based on a single criteria (i.e. the number of hours) for sake of simplicity, they can be extended to the multicriteria case, where the preference is expressed as a tuple of incomparable criteria.

Example 4.1 [Tell and negotiation] P_1 and P_2 both want to present their policy

(respectively represented by c_4 and c_3) to the other party and to find a shared agreement on the service (i.e. a SLA). Their agent description is: $P_1 \equiv \langle tell(c_4) \rightarrow^0_\infty tell(s_{p2}) \rightarrow^0_\infty ask(s_{p1}) \rightarrow^2_{10} success\rangle || \langle tell(c_3) \rightarrow^0_\infty tell(s_{p1}) \rightarrow^0_\infty ask(s_{p2}) \rightarrow^1_4 success\rangle \equiv P_2$, executed in the store with empty support (i.e. $\bar{0}$). Variables s_{p1} and s_{p2} are used only for synchronization and thus will be ignored in the following considerations (e.g. replaced by the $SYNCHRO_i$ agents in Ex. 4.2). The final store (the merge of the two policies) is $\sigma = (c_4 \otimes c_3) \equiv 2x + x + 5$, and since $\sigma \Downarrow_\emptyset = 5$ is not included in the last preference interval of P_2 (between 1 and 4), P_2 does not succeed and a shared agreement cannot be found. The practical reason is that the failure management systems of P_1 need at least 5 hours (i.e. $c_4 = x + 5$) even if no failures happen (i.e. $x = 0$). Notice that the last interval of P_2 requires that at least 1 hour is spent to check failures.

Example 4.2 [Retract] After some time (still considering Ex. 4.1), suppose that P_1 wants to relax the store, because its policy is changed: this change can be performed from an interactive console or by embedding timing mechanisms in the language as explained in [4]. The removal is accomplished by retracting c_1, which means that P_1 has improved its failure management systems. Notice that c_1 has not ever been added to the store before, so this retraction behaves as a relaxation; partial removal, which cannot be performed with tokens (see Sec. 5), is clearly important in a negotiation process. $P_1 \equiv \langle tell(c_4) \rightarrow^0_\infty SYNCHRO_{P1} \rightarrow^2_{10} retract(c_1) \rightarrow^2_{10} success\rangle || \langle tell(c_3) \rightarrow^0_\infty SYNCHRO_{P2} \rightarrow^1_4 success\rangle \equiv P_2$ is executed in $\bar{0}$. The final store is $\sigma = c_4 \otimes c_3 \ominus c_1 \equiv 2x + 2$, and since $\sigma \Downarrow_\emptyset = 2$, both P_1 and P_2 now succeed (it is included in both intervals).

Example 4.3 [Nask] In a negotiation scenario, the *nask* operation can be used for several purposes. Since it checks the absence of information (see Sec. 3), for example it can be used to check if the own policy is still implied by the store or if it has been relaxed too much: e.g. $P_1 \equiv \langle retract(c_1) \rightarrow^0_\infty SYNCHRO_{P1} \rightarrow^0_\infty success\rangle || \langle tell(c_4) \rightarrow^0_\infty nask(c_4) \rightarrow^0_\infty tell(c_4) \rightarrow^0_\infty SYNCHRO_{P2} \rightarrow^0_\infty success\rangle \equiv P_2$ (evaluated in $\bar{0}$). As soon as P_2 adds its policy (i.e. c_4), P_1 can relax it (by removing c_1); P_1 perceives this relaxation with the *nask* and adds again c_4. The reason is that P_1 explicitly needs a global number of spent hours not better than that one defined by c_4, which then must be entailed by the store: e.g. its recovery system works only with at least that time. Here the preference intervals of the two agents are not significative, since equal to the whole $\mathbb{R}^+$.

Example 4.4 [Update]The *update* can be instead used for substantial changes of the policy: for example, suppose that $P_1 \equiv \langle tell(c_1) \rightarrow^0_\infty update_{\{x\}}(c_6) \rightarrow^0_\infty success, \bar{0}\rangle$. This agent succeeds in the store $\bar{0} \otimes c_1 \Downarrow_{(V \setminus \{x\})} \otimes c_6$, where $c_1 \Downarrow_{(V \setminus \{x\})} = \bar{3}$ and $\bar{3} \otimes c_6 \equiv y + 4$ (i.e. the polynomial describing the final store). Therefore, the first policy based on the number of failures (i.e. c_1) is updated such that x is "refreshed" and the new added policy (i.e. c_6) depends only on the y number of system reboots. The consistency level of the store (i.e. the number of hours) now depends only on the y variable of the SCSP. Notice that the $\bar{3}$ component of the final store derives from the "old" c_1, meaning that some fixed management delays

are included also in this new policy.

5 Related Work

Nonmonotonicity has been extensively studied for crisp constraints in the so-called *linear cc* programming [17] and in following works as [1,9,11,16]. Regarding related SLA negotiation models, the process calculus introduced in [12] is focused on controlling and coordinating distributed process interactions while respecting QoS parameters expressed as c-semiring values; however, the model does not cover negotiation. In [2] and [15] the authors define SLAs at a lower level of abstraction and their description is separated from their negotiation (while soft constraint systems cover both cases).

The most direct comparison for *nmsccp*, since the two languages are both used for SLA negotiation, is with the work in [8], in which soft constraints are combined with a name-passing calculus (even if all the examples in the paper are then developed using crisp constraints). However, w.r.t our language there are some important differences: *i)* in *nmsccp* we do not have the concept of constraint token and it is possible to remove every c that is entailed by the store (i.e. $\sigma \sqsubseteq c$), even if c is syntactically different from all the c previously added (as the retraction of c_1 in Ex. 4.2). For example, even the removal of the $c_1 \otimes c_2$ composition from a store containing both c_1 and c_2 cannot be performed in [8], because it is a derived constraint. Therefore our *retract* is more like a "relaxation" operation, and not a "physical" removal of a token as in [8]; this feature is in the nature of negotiation, when a step back must be taken to reach a shared agreement.

Then, *ii)* with *nmsccp* we can reach a final agreement among the parties, knowing also "how consistently" (or "how expensively") the claimed needs are being satisfied. This is accomplished by checking the preference level of the store and the consistency intervals conditioning the actions (Fig. 2). In this way, each of the agents can specify its desired preference for the final agreement. This is a relevant improvement with regard to [8], where the final store collects all the consistent solutions without any distinction, i.e. each solution that satisfies $\sigma \Downarrow_{\emptyset} = \alpha_i$, for every $\alpha_i >_S \mathbf{0}$.

At last, *iii)* we introduced the *update* operation (extending the semantics of the crisp *update* in [11]), which is a variable-grained relaxation, and the *nask* (whose crisp version is in [11]), that is very useful to have in a nonmonotonic framework to check absence of information. Notice that we do not need the *check* operation defined in [8] in order to verify if a given constraint is consistent with the store (without adding it). The reason is that we have the checked transitions of Fig. 2 to prevent the store from becoming not consistent "enough".

6 Conclusions and Future Work

Monotonicity is one the major drawbacks for practical use of concurrent constraint languages in reactive and open systems. In this paper we have proposed some new primitives (*nask*, *update* and *retract*) that allow the nonmonotonic evolution of the

store. We have chosen to extend *sccp* because soft constraints [3,6] enhance the classical constraints in order to represent consistency levels, and to provide a way to express preferences, fuzziness, and uncertainty. We think that having preference values directly embedded in the language represents a valuable solution to manage SLA negotiation, particularly when a given QoS is associated with the resources. Soft constraints can be used to model different problems by only parameterizing the semiring structure.

We would like to merge this language with the timing mechanisms (e.g. "time-out" and "interrupt") explained in [4]. These capabilities can be useful during complex interactions, e.g. to interrupt a long wait for pending conditions (or to interrupt a deadlock) or to trigger urgent actions.

Moreover, we would like to investigate the possibility of a distributed store instead of the centralized one we have assumed in this paper. In distributed CSP [21], variables and constraints are distributed among all the agents, thus the knowledge of the problem is not concentrated in a single agent only. This requirement is common in many practical application, and surely for (SLA) negotiating entities, where each agent has a private store collecting its resources (i.e. variables) and policies (i.e. constraints).

At last, we plan to provide the language with other formal tools, such as a denotational semantics, a study on agent equivalences in order to prove when two providers offer the same service. Moreover, we want to deepen the absence of failures in *nmsccp*.

References

[1] Best, E., F. S. de Boer and C. Palamidessi, *Partial Order and SOS Semantics for Linear Constraint Programs*, in: *COORDINATION'97*, LNCS **1282** (1997), pp. 256–273.

[2] Bhoj, P., S. Singhal and S. Chutani, *SLA management in federated environments*, Comput. Networks **35** (2001), pp. 5–24.

[3] Bistarelli, S., "Semirings for Soft Constraint Solving and Programming," LNCS **2962**, Springer, 2004.

[4] Bistarelli, S., M. Gabbrielli, M. C. Meo and F. Santini, *Timed Soft Concurrent Constraint Programs*, in: D. Lea and G. Zavattaro, editors, *COORDINATION'08*, LNCS **5052** (2008), pp. 50–66.

[5] Bistarelli, S. and F. Gadducci, *Enhancing Constraints Manipulation in Semiring-Based Formalisms*, in: *ECAI'06* (2006), pp. 63–67.

[6] Bistarelli, S., U. Montanari and F. Rossi, *Semiring-based constraint satisfaction and optimization*, J. ACM **44** (1997), pp. 201–236.

[7] Bistarelli, S., U. Montanari and F. Rossi, *Soft concurrent constraint programming*, ACM Trans. Comput. Logic **7** (2006), pp. 563–589.

[8] Buscemi, M. G. and U. Montanari, *CC-Pi: A constraint-based language for specifying service level agreements*, in: *ESOP'07*, LNCS **4421** (2007), pp. 18–32.

[9] Codognet, P. and F. Rossi, *NMCC programming: Constraint enforcement and retracting in CC programming*, in: *ICLP'95*, 1995, pp. 417–431.

[10] Cohen, D., M. Cooper, P. Jeavons and A. Krokhin, *The complexity of soft constraint satisfaction*, Artif. Intell. **170** (2006), pp. 983–1016.

[11] de Boer, F. S., J. N. Kok, C. Palamidessi and J. J. M. M. Rutten, *Non-monotonic Concurrent Constraint Programming*, in: *ILPS'93*, 1993, pp. 315–334.

[12] De Nicola, R., G. L. Ferrari, U. Montanari, R. Pugliese and E. Tuosto, *A process calculus for QoS-aware applications*, in: *COORDINATION'95*, LNCS **3454** (2005), pp. 33–48.

[13] Fages, F., P. Ruet and S. Soliman, *Linear concurrent constraint programming: operational and phase semantics*, Inf. Comput. **165** (2001), pp. 14–41.

[14] Jennings, N., P. Faratin, A. Lomuscio, S. Parsons, C. Sierra and M. Wooldridge, *Automated negotiation: prospects, methods and challenges*, Int. Journal of Group Decision and Negotiation **10** (2001), pp. 199–215.

[15] Keller, A. and H. Ludwig, *The WSLA framework: Specifying and monitoring service level agreements for web services*, J. Netw. Syst. Manage. **11** (2003), pp. 57–81.

[16] Saraswat, V., R. Jagadeesan and V. Gupta, *Default timed concurrent constraint programming*, in: *POPL'95* (1995), pp. 272–285.

[17] Saraswat, V. and P. Lincoln, *Higher-order Linear Concurrent Constraint Programming*, Technical report, Xerox Parc (1992).

[18] Saraswat, V. and M. Rinard, *Concurrent constraint programming*, in: *POPL'90* (1990), pp. 232–245.

[19] Saraswat, V. A., R. Jagadeesan and V. Gupta, *Default Timed Concurrent Constraint Programming*, in: *POPL'95* (1995), pp. 272–285.

[20] Saraswat, V. A., M. Rinard and P. Panangaden, *The semantic foundations of concurrent constraint programming*, in: *POPL'91* (1991), pp. 333–352.

[21] Yokoo, M., E. H. Durfee, T. Ishida and K. Kuwabara, *The distributed constraint satisfaction problem: Formalization and algorithms*, IEEE Transactions on Knowledge and Data Engineering **10** (1998), pp. 673–685.