Designing a Nonmonotonic Soft Concurrent Constraint Language for SLA Management

Stefano Bistarelli[1,2] and Francesco Santini[2,3]

[1] Dipartimento di Scienze, Università "G. D'Annunzio" di Chieti-Pescara, Italy
bista@sci.unich.it
[2] Istituto di Informatica e Telematica (CNR), Pisa, Italy
stefano.bistarelli@iit.cnr.it,francesco.santini@iit.cnr.it
[3] IMT - Institute for Advanced Studies, Lucca, Italy
f.santini@imtlucca.it

Abstract. We present an extension of the *Soft Concurrent Constraint* language to allow 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 $\mathit{update}_X(c)$ transactionally relaxes all the constraints of the store that deal with the variables in X set, and then adds a constraint c (usually with $\mathit{support} = X$); 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. The main advantage is to have a preference (or cost) measure directly embedded in the language, and to have a highly flexible and parametric abstraction.

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 [13] 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 [13] . Notice that this process is continuous because clients and providers can change their requirements during their execution, and the same QoS can be improved or degraded for many reasons (e.g. due to the system load).

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" w.r.t. the threshold. Since constraints can only be accumulated (via the *tell* operation), the consistency level of the store can only monotonically decrease starting from the initial empty store. In fact, the function used to combine together 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 w.r.t. 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 much 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 w.r.t. 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) [2,14] 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 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 w.r.t. 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 . Section 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 concludes by indicating future research directions.

2 Background

Absorptive Semiring. An absorptive semiring [5] can be represented as a tuple $\langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$ 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. The absorptive property is due to the fact that the semiring is commutative and $\mathbf{1}$ is the unit element for $+$ (i.e. $a + \mathbf{1} = \mathbf{1}$). A c-semiring is an absorptive semiring $\langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$ such that: $+$ is idempotent, $\mathbf{1}$ is its absorbing element and $\times$ is commutative. Let us consider the relation $\leq_S$ over A such that $a \leq_S b$ iff $a + b = b$. Then it is possible to prove that (see [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*. Moreover, in [5] the authors extended the semiring structure by adding the notion of *division*, i.e. $\div$, as a weak inverse operation of $\times$. For a full explanation and properties of $\div$, please refer to [5].

Definition 1 ([5]). *Let $\mathcal{K}$ be an absorptive semiring. Then*

- *$\mathcal{K}$ is* invertible *if for all elements $a, b \in A$ such that $a \leq b$ there exists an element $c \in A$ such that $b \times c = a$;*
- *it is* weakly uniquely *invertible if c is unique whenever $a < b$;*
- *it is* uniquely *invertible if c is unique whenever $b \neq \mathbf{0}$.*

Definition 2 ([5]). *Let $\mathcal{K}$ be an absorptive, invertible semiring. Then, $\mathcal{K}$ 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 $a \leq b$.*

Definition 3 ([5]). *Let $\mathcal{K}$ be an absorptive semiring. Then, $\mathcal{K}$ 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. Then, the following theorem can be proved:

Theorem 1 ([5]). *Let $\mathcal{K}$ be a absorptive semiring. If $\mathcal{K}$ is complete*[4]*, then it is residuated. 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.*

[4] If $\mathcal{K}$ is a absorptive semiring, then $\mathcal{K}$ is complete if it is closed with respect to infinite sums, and the distributivity law holds also for an infinite number of summands.

Constraint System. A *soft constraint* [6,3] may be seen as a constraint where each instantiation of its variables has an associated preference. Given a semiring $S = \langle A, +, \times, \mathbf{0}, \mathbf{1}\rangle$ and an ordered set of variables V over a finite domain D, a soft constraint is a function which, given an assignment $\eta : V \to D$ of the variables, returns a value of the semiring. Using this notation $\mathcal{C} = \eta \to A$ 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 $c_{x,y} : V \to D \to A$, 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 $v := d_1$. Note also that $c\eta$ is the application of a constraint function $c : V \to D \to A$ to a function $\eta : V \to D$; what we obtain, is a semiring value $c\eta = a$.

Given the set $\mathcal{C}$, the combination function $\otimes : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ 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 $\ominus : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ is instead defined as $(c_1 \ominus c_2)\eta = c_1\eta \div c_2\eta$ [5]. Informally, performing the $\otimes$ or the $\ominus$ 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 $V - \{v\}$, written $c \Downarrow_{(V-\{v\})}$ 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 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 $a \neq b$. Now it is possible to define a constraint systems *"a la Saraswat"* [7]: notice that in *sccp*, algebricity is not required, since the algebraic nature of $\mathcal{C}$ strictly depends on the properties of the semiring [7]:

Theorem 2 (cylindric system [7]). *Consider a semiring $S = \langle A, +, \times, \mathbf{0}, \mathbf{1}\rangle$, a domain of the variables D, an ordered set of variables V, the corresponding structure $\mathcal{C}$. Then, $S_C = \langle \mathcal{C}, \otimes, \bar{\mathbf{0}}, \bar{\mathbf{1}}, \exists_x, d_{xy}\rangle$[5], is a cylindric constraint system.*

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

3 The Language

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. 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 can be removed, even if c is different from any other constraints previously added to the store 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 X set, and then adds a constraint c (usually with $support = X$). This operation is variable-grained w.r.t. our *retract*, and for many applications (as ours, on SLA negotiation), it is very convenient to have a remove 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).

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 already 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.

The Syntax of the Language. Given a soft constraint system as defined in Theorem 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.

$$
\begin{array}{rl}
P ::= & F.A \\
F ::= & p(Y) :: A \mid F.F \\
A ::= & success \mid tell(c) \rightarrowtail A \mid retract(c) \rightarrowtail A \mid update_X(c) \rightarrowtail A \mid E \mid A \| A \mid \exists x.A \mid p(Y) \\
E ::= & ask(c) \rightarrowtail A \mid nask(c) \rightarrowtail A \mid E + E
\end{array}
$$

Fig. 1. Syntax of the nmsccp language.

In addition to the new operations, the other most important variation w.r.t. *sccp* is the action prefixing symbol $\rightarrowtail$ in the syntax notation, which can be considered as a general "checked" transition of the type $\rightarrow^{\varphi_2}_{\varphi_1}$, where $\varphi_i = a_i$ (i.e. the threshold is a semiring element) or $\varphi_i = \phi_i$ (i.e. the threshold is a constraint) with $i = 1, 2$. In words, two conditions must be checked at the same time: a_1 or ϕ_1 (one of the two) 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 much 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). 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.

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.

Moreover, $\rightarrowtail$ simplifies the writing of the language syntax (Fig. 1) and operational semantics rules (Fig. 3), by avoiding to replicate the same rules differentiating among themselves only on the a or ϕ check (as *tell*/valued-*tell* and *ask*/valued-*ask* in *sccp* [7]). Notice that the classical *ask* and *tell* operations in *sccp* (where only the lower bound is present) are preserved also in *nmsccp*: e.g. $ask/tell(c) \rightarrow^{\bar{1}}_{\phi} A$ represent classical *sccp ask*/*tell*, and $ask/tell(c) \rightarrow^{\bar{1}}_{a}$ represent their *sccp* valued versions (with $\bar{\mathbf{1}}$ defined as in the footnote of Theorem 2).

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 $\sigma \in C$ while the set of terminal configurations is instead $T = \{\langle \mathit{success}, \sigma \rangle\}$. 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 $\mathit{check}_{\rightarrowtail} : \sigma \rightarrow \{\mathit{true}, \mathit{false}\}$ (where $\sigma \in C$), 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 cannot clearly be "better" than the upper one, otherwise the condition is intrinsically wrong.

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

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

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

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

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

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 $\not\sqsubseteq$ 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 od the problem variables). This is called the *best level of consistency* and it is defined by $\mathit{blevel}(P) = \mathit{Sol}(P) \Downarrow_{\emptyset}$, where $\mathit{Sol}(P) = (\bigotimes C) \Downarrow_{con}$; notice that $\mathit{supp}(\mathit{blevel}(P)) = \emptyset$. We also say that: P is α-consistent if $\mathit{blevel}(P) = \alpha$; 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, for example, **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 much 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.

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

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

$$\textbf{R3}\ \frac{\langle A, \sigma\rangle \longrightarrow \langle A', \sigma'\rangle}{\begin{array}{c}\langle A \parallel B, \sigma\rangle \longrightarrow \langle A' \parallel B, \sigma'\rangle\\ \langle B \parallel A, \sigma\rangle \longrightarrow \langle B \parallel A', \sigma'\rangle\end{array}}\quad \textbf{Parall1}$$

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

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

$$\textbf{R6}\ \frac{\sigma \nvdash c \quad check(\sigma)_{\rightarrowtail}}{\langle nask(c) \rightarrowtail A, \sigma\rangle \longrightarrow \langle A, \sigma\rangle}\quad \textbf{Nask}$$

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

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

$$\textbf{R9}\ \frac{\langle A[x/y], \sigma\rangle \longrightarrow \langle B, \sigma'\rangle}{\langle \exists x.A, \sigma\rangle \longrightarrow \langle B, \sigma'\rangle}\ \text{with } y \textit{ fresh}\quad \textbf{Hide}$$

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

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. Notice that 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 w.r.t. the lower and upper thresholds set by the programmer, defined by the specific $\rightarrowtail$ transition arrow: i.e. if $check(\sigma)_{\rightarrowtail}$ is true.

Parallelism and nondeterminism: the composition operators + and $\parallel$ are not modified w.r.t. [7]. A parallel agent (rules **R3** and **R4**) will succeed when all the agents succeed the parallel. 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 non-determinism.

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 $\ominus$ 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 three 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 $c_2 \ominus c_1$ 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 $c_3 \ominus c_1$ 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. Finally, **R7** requires that the consistency interval satisfies $\sigma' = \sigma \ominus c$. Clearly, it is also possible to completely remove a constraint as if using tokens: i.e. $\langle tell(c_i) \rightarrowtail retract(c_i) \rightarrowtail A, \sigma_k \rangle$ is equivalent (for every c_i, σ_k and $\rightarrowtail$ if enabled) to $\langle A, \sigma_k \rangle$ due to the properties explained in [5], i.e. $a \times b \div b = a$ always holds (where a and b are any two values of the semiring set).

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

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

Fig. 4. Four weighted soft constraints c_1, c_2, c_3 and c_4, where $c_4 = c_2 \ominus c_1$.

The semantics of **Update** rule (**R8**) 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 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. $\sigma \Downarrow_{V-x} = \sum_{d_i \in D} c\eta[x := d_i]$, 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.

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**) is not modified w.r.t. the classical one: as usual, we use the notion of diagonal constraints (as defined in Sec. 2) 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*): $\mathcal{S}_A = \{\sigma \Downarrow_{var(A)} | \langle A, \bar{\mathbf{1}} \rangle \rightarrow^* \langle success, \sigma \rangle\}$

A First Example. To exemplify the rules in Fig. 3, now we show an example where we evaluate a *nmsccp* agent in the starting empty store $\bar{\mathbf{1}}$ (i.e. the store with empty support). All the checked transitions of the agent are of the type **C1** in Fig. 2. The weighted constraints c_1 and c_2 of this example are those represented in Fig. 4, and the chosen semiring is still the *Weighted* one: $\langle \mathbb{R}^+, min, +, +\infty, 0 \rangle$, where $+$ is the arithmetic operator. The agent and the initial store are represented in (1). According to rule **R1**, the store then becomes $\bar{0} \otimes c_2 = c_2$ because $+\infty \not< c_2 \Downarrow_\emptyset \not< 8$ is true (refer to **C1**); the reason is that $c_2 \Downarrow_\emptyset = 8$, when $x = 0$. Thus the first *tell* succeeds and the agent reach the state in (2). To execute the *ask*, according to **R2** and **C1**, we must check if both c_1 is entailed by the store (i.e. if $c_2 \vdash c_1$) and $18 \not< c_2 \Downarrow_\emptyset \not< 7$. These two conditions are satisfied because, respectively, $c_2 \sqsubseteq c_1$ and $c_2 \Downarrow_\emptyset = 8$. Thus, the state in (3) is reached; here, the first precondition of rule **R7** (i.e. of the *retract*) is true since $c_2 \sqsubseteq c_1$. Thus, the agent can retract c_1 from the store (i.e. c_2): the resulting store is c_4 (defined in Fig. 4) because the instantiation of $\div$ for *Weighted* semirings [5] is the arithmetic difference between two semiring values a and b, i.e. a $\div b \equiv a - b$ if $a > b$, or 0 otherwise. Following the definition of constraint division given in Sec. 2, for every η assignment of x (the domain of x is $\mathbb{N}$), we have $(c_2 \ominus c_1)\eta = c_2\eta \div c_1\eta = (2x + 8) - (x + 3) = x + 5 \equiv c_4$. Notice that c_4 satisfies the second precondition of **R7** (concerning **C1**) since $18 \not< c_4 \Downarrow_\emptyset \not< 3$ (when $x = 0$, then $c_4(x) = 5$ and $18 \not< 5 \not< 3$), and then the agent can execute the *retract* action becoming the *success* agent in (4).

$$\langle tell(c_2) \rightarrow^{8}_{+\infty} ask(c_1) \rightarrow^{7}_{18} retract(c_1) \rightarrow^{3}_{18} success, \bar{0} \rangle \tag{1}$$

$$\langle ask(c_1) \rightarrow^{7}_{18} retract(c_1) \rightarrow^{3}_{18} success, c_2 \rangle \tag{2}$$

$$\langle retract(c_1) \rightarrow^{3}_{18} success, c_2 \rangle \tag{3}$$

$$\langle success, c_4 \rangle \tag{4}$$

Difference between retract *and* update *operations.* We would like to highlight that the *retract* and *update* consist in two quite different actions and some important benefits derive from including both these primitives: the *update* is large-grained from the constraint point of view but, at the same time, is fine-grained from the variables point of view; the *retract* is instead fine-grained for what concerns the constraint relaxation, but cannot directly deal with variables. With the next very simple example we clarify this concept by using the c_1 constraint in Fig. 4. Starting under the same conditions, the two actions show a different behaviour (final states (7) and (10)): the reason is that in (10) the best semiring value depending on x for c_1 is found (i.e. $\sigma \Downarrow_x = \sum_{d_i \in D} c\eta[x := d_i]$. Therefore, c_1 continues to be somewhat present in the store.

$$\langle tell(c_1) \rightarrow^0_{+\infty} retract(c_1) \rightarrow^0_{+\infty} success, \bar{0}\rangle \quad (5)$$
$$\langle retract(c_1) \rightarrow^0_{+\infty} success, c_1\rangle \quad (6)$$
$$\langle success, \bar{0}\rangle \quad (7)$$

$$\langle tell(c_1) \rightarrow^0_{+\infty} update_x(\bar{0}) \rightarrow^0_{+\infty} success, \bar{0}\rangle \quad (8)$$
$$\langle update_x(\bar{0}) \rightarrow^0_{+\infty} success, c_1\rangle \quad (9)$$
$$\langle success, \bar{5}\rangle \quad (10)$$

No Failures. One fundamental property of the agents behaviour can be spotted: their computations 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 w.r.t. *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 (starting from the initial store $\bar{\mathbf{1}}$), 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, 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 finitary.

As a practical example of (a specific subset of) soft constraints that have a finitary 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 $\{V - y\}$ (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 obtaining x^2 as result (in the *Weighted* semiring, to maximize the preference means to minimize the polynomial).

Otherwise, if soft constraints have not a finitary 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 $c_1 \otimes c_2 \ominus c_3 \Downarrow_{\{V-X\}} \otimes c4$ (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. In *nmsccp*, the actions ordering is important when dealing with an idempotent $\times$: e.g. $\langle tell(c_1) \rightarrowtail retract(c_1) \rightarrowtail tell(c_1) \rightarrowtail A, \bar{\mathbf{1}}\rangle \equiv \langle A, c_1\rangle$ but $\langle tell(c_1) \rightarrowtail tell(c_1) \rightarrowtail retract(c_1) \rightarrowtail A, \bar{\mathbf{1}}\rangle \equiv \langle A, \bar{\mathbf{1}}\rangle$ if $\times$ is idempotent. 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 Agreement

One of the most meaningful application of the *nmsccp* language is on the modelling procedure of generic entities negotiating a formal agreement, i.e. a SLA. The main task consists in accomplishing the requests of all the agents by satisfying the needs of everybody and then allocating the resources. The agreement is reached by considering multiple QoS indicators, as fault tolerance, availability, scalability, time performance or other attributes of the service like billing.

The example we describe here (the program is in Fig. 6) is based on a negotiation with an associated preference between a provider **P** and a client **C** asking for a dynamic internet connection. The variables of our SCSP are x and y, which respectively represent the bandwidth and the latency of the connection measured on Mbps and msec (the domain of both x and y is $\mathbb{R}+$).

Since the cartesian product of multiple c-semirings is still c-semiring [3], this can be fruitfully used to have a multi-criteria negotiation. Our first cost is represented by the money cost at which the service is sold: thus we use the classical *Weighted* semiring (see Sec. 3.1). The second QoS feature is availability, i.e. the probability that the service is up. Therefore, the *Probabilistic* semiring is chosen: $\langle [0, 1], max, \times, 0, 1\rangle$; $a \div b$ is defined as $\frac{a}{b}$ if $a < b$ (i.e. the arithmetic division) or as 1 otherwise [5]. Their composition produces the semiring used in the example: $S_{Neg} = \langle\langle \mathbb{R}^+, [0, 1]\rangle, \langle min, max\rangle, \langle +, \times\rangle, \langle +\infty, 0\rangle, \langle 0, 1\rangle\rangle$. Therefore, any preference value (related to the store, constraints and thresholds) now corresponds to a pair given by $\langle money, availability\rangle$ (i.e. $\langle €, \%\rangle$). Moreover, the $+$ and $\times$ of the semiring are now vectorized and deal with pairs of values.

Some of the constraints used in the example in Fig. 6 are described in Fig. 5 (i.e. c_1, c_2 and c_3) just to give the idea; c_4 (Fig. 6) is defined as $c_4 = c_1 \otimes c_2$ and it is used as an alias name (a shortcut) for the initial offer of **P**. In words, c_1 constraints only bandwidth (i.e., x) and has the fixed cost $\langle 5, 0.98\rangle$ when $1Mbps \leq x \leq 4Mbps$, and then, as the bandwidth increases (i.e. $4 < x \leq 8$), the cost worsens according to the two polynomials $\langle 2x, \frac{x-8}{4-8}\rangle$; c_2 binds together the preferences for x and y, while c_3 represents the initial requirements of **C** about money and availability:

$$c_1 : \{x\} \to \mathbb{R}^+ \to \langle \mathbb{R}^+, [0,1], \mathbb{R}^+ \rangle \quad \text{s.t. } c_1(x) = \begin{cases} \langle 5, 0.98 \rangle & \text{if } 1 \le x \le 4, \\ \langle 2x, \frac{x-8}{4-8} \rangle & \text{if } 4 < x \le 8, \\ \langle +\infty, 0 \rangle & \text{otherwise.} \end{cases}$$

$$c_2 : \{x, y\} \to \mathbb{R}^+ \times \mathbb{R}^+ \to \langle \mathbb{R}^+, [0,1], \mathbb{R}^+ \rangle \quad \text{s.t. } c_2(x,y) = \begin{cases} \langle 1, 0.99 \rangle & \text{if } (1 \le x \le 8) \wedge (150 < y \le 200), \\ \langle 4, 0.97 \rangle & \text{if } (1 \le x \le 4) \wedge (100 \le y \le 150), \\ \langle 10, 0.96 \rangle & \text{if } (4 < x \le 8) \wedge (100 \le y \le 150), \\ \langle +\infty, 0 \rangle & \text{otherwise.} \end{cases}$$

$$c_3 : \{x\} \to \mathbb{R}^+ \to \langle \mathbb{R}^+, [0,1], \mathbb{R}^+ \rangle \quad \text{s.t. } c_3(x) = \langle 4, 0.99 \rangle \quad \text{if } 1 \le x \le 4$$

Fig. 5. Some of the constraints used in Fig. 6 (x is bandwidth and y is latency).

C is only interested in having a connection with a bandwidth between 1Mbps and 4Mbps with an availability $\not< 0.99\%$ and a cost $\not> 4€$ (no concern about y).

The **Negotiation and Acquisition** interaction is described in Fig. 6 and is given by the parallelization of **P** and **C** starting from the store with empty support $\langle \bar{0}, \bar{1} \rangle$. **P** makes its initial offer about the service, by adding $c_4 = c_1 \otimes c_2$ to the store. It is the seller and must be the first proposer. Clearly, for each pair of $\langle x, y \rangle$ domain values, the global cost of this offer is given by summing together the money-costs and multiplying together the availability values defined by c_1 and c_2: e.g. for $(1 \le x \le 4) \wedge (150 < y \le 200)$ the cost is $\langle 5+1, 0.98 \times 0.99 \rangle = \langle 6, 0.9702 \rangle$. The checked transitions of the example in Fig. 6 are all of **C1** type (see Fig. 2) and represent the preference interval requested by the actions; for sake of example simplicity, all the **P** actions are labeled with the same interval, and same consideration applies to **C** actions: $\dot{\rightarrowtail} \equiv \rightarrow^{\langle 5,0.99 \rangle}_{\langle \infty,0 \rangle}$ means **P** does not want to sell with a price better than $\langle 5, 0.99 \rangle$, and $\ddot{\rightarrowtail} \equiv \rightarrow^{\langle 0,1 \rangle}_{\langle 20,0.9 \rangle}$ means **C** does not want to buy at a price worse than $\langle 20, 0.9 \rangle$. From Fig. 2, we remind that these checks are on the best possible solution of the store.

In Fig. 6, **C** waits for presence of the **P** offer in the store (i.e., $ask(c_4)$); we can consider c_4 as a constraint global name, thus **C** knows the name without having the details of the constraint (belonging to **P**). Then, **C** checks whether its requirements are satisfied or not ($nask(c_3) + ask(c_3)$). From Fig. 5 we can see that $c_4 \sqsubseteq c_3$ (with $\sigma = c_4$) and then $ask(c_3)$ is enabled, since when $1 \le x \le 4$ then $\langle 5, 0.98 \rangle <_{S_{Neg}} \langle 4, 0.99 \rangle$; this means that **P** offers conditions worse than those demanded by **C**, i.e. **P** offers a higher cost and a lower availability for the requested bandwidth. Therefore, **C** needs to make a counter-offer to reach a shared agreement, and it relaxes the **P** offer (i.e. $retract(c_5)$, c_5 is left generic) to ask for a better treatment. Otherwise, if $nask(c_3)$ had been enabled instead, **C** needs to refine its request through the *REFINE* agent, since some of its requirements are met but some may be not (the *nask* is enabled when the $\sigma \not\vdash c$, but we do not know if $c \sqsubseteq \sigma$). Then it can buy the service. Describing **P** in Fig. 6, it checks if **C** wants to immediately buy (i.e. $ask(c_7)$) or if the store has been relaxed (i.e. $nask(c_4)$); in this case, **P** checks if an acceptable cost is still implied by the store (i.e. if $ask(c_6)$ is enabled), otherwise it makes a counter-counter-offer (with $tell(c_8)$). The c_6 and c_8 are left generic but can be considered similar to those in Fig. 5.

SYNCHRO$_C$ and *SYNCHRO$_P$* are dual agents that are used to synchronize **P** and **C** on the end of the renegotiation after the $retract(c_5)$ and $tell(c_8)$ actions, and also to perform a final check on the store before buying and selling; synchronization could be achieved via a synchronization variable. *USE* and *MONITOR* are generic agents that model the use and the supervision of the resources, while the presence of c_7 in the store implies the acquisition of the service, i.e. it is a synchronization variable (its name is shared between **P** and **C**, as c_4). All these agents, as well as some of used constraints, are not detailed in order to have a more intuitive program.

$$\begin{gathered}\mathbf{P} ::- tell(c_4) \dot{\rightarrowtail} (ask(c_7) \dot{\rightarrowtail} sell() + nask(c_4) \dot{\rightarrowtail} (ask(c_6) \dot{\rightarrowtail} SYNCHRO_P \dot{\rightarrowtail} sell() + \\ nask(c_6) \dot{\rightarrowtail} tell(c_8) \dot{\rightarrowtail} SYNCHRO_P \dot{\rightarrowtail} sell()))\end{gathered}$$

$$\begin{gathered}\mathbf{C} ::- ask(c_4) \ddot{\rightarrowtail} (nask(c_3) \ddot{\rightarrowtail} REFINE \ddot{\rightarrowtail} buy() + ask(c_3) \ddot{\rightarrowtail} retract(c_5) \ddot{\rightarrowtail} SYNCHRO_C \\ \ddot{\rightarrowtail} buy())\end{gathered}$$

$$\mathbf{sell()} :: ask(c_7) \dot{\rightarrowtail} MONITOR \dot{\rightarrowtail} nask(c_7) \dot{\rightarrowtail} success$$

$$\mathbf{buy()} :: tell(c_7) \ddot{\rightarrowtail} USE \ddot{\rightarrowtail} retract(c_7) \ddot{\rightarrowtail} success$$

$$\textbf{Negotiation and Acquisition} ::- \langle Provider \parallel Client,\ \langle \bar{0}, \bar{1} \rangle \rangle$$

Fig. 6. Negotiating and acquiring an internet connection by using *nmsccp* agents. **P** is the provider agent and **C** is the client. The c_4 constraint is an alias name for the initial offer **P**, i.e. $c_4 = c_1 \otimes c_2$. Moreover, $\dot{\rightarrowtail} \equiv \rightarrow^{\langle 5,0.99\rangle}_{\langle\infty,0\rangle}$ and $\ddot{\rightarrowtail} \equiv \rightarrow^{\langle 0,1\rangle}_{\langle 20,0.9\rangle}$.

4.1 Related Work

Nonmonotonicity has been extensively studied for crisp constraints in the so-called *linear cc* programming [17] and in following works as [9, 11, 16, 1]. 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. *SLAng* [15] and *WSLA* [14] are XML-based languages for defining SLAs, therefore, at a lower level of abstraction.

The most direct comparison for *nmsccp*, since the the two languages are 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. 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 relaxation feature is in the nature of negotiation, when a step back must be taken to reach a shared agreement. When having an idempotent $\times$, a further difference w.r.t. the *retract* semantics in [8] is explained in the last paragraph of Sec. 3.1, describing the preference representation.

Then, *ii)* with *nmsccp* we can model the negotiation procedure and 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 w.r.t. [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 which is a variable-grained relaxation, and the *nask*, 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".

5 Conclusions and Future Work

Monotonicity is one the mayor 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 are currently extending the language with timing mechanisms as "timeout" and "interrupt" to further improve the expressiveness of the language [4]. These capabilities can be useful during complex interactions, e.g. to interrupt a long wait for pending conditions or to trigger some urgent and critical actions.

At last, we plan to provide the language with other formal tools, as a denotational semantics, a study on agent equivalences in order to prove when two providers offer the same service.

References

1. E. Best, F. S. de Boer, and C. Palamidessi. Partial order and sos semantics for linear constraint programs. In *COORDINATION '97: Proceedings of Coordination Languages*

and Models, pages 256–273, London, UK, 1997. Springer-Verlag.
2. P. Bhoj, S. Singhal, and S. Chutani. SLA management in federated environments. *Comput. Networks*, 35(1):5–24, 2001.
3. S. Bistarelli. *Semirings for Soft Constraint Solving and Programming*, volume 2962 of *Lecture Notes in Computer Science*. Springer, 2004.
4. S. Bistarelli, M. Gabrielli, M. C. Meo, and F. Santini. Timed concurrent constraint programs. In *Doctoral Program Informal Proceedings, CP'07*, 2007.
5. S. Bistarelli and F. Gadducci. Enhancing constraints manipulation in semiring-based formalisms. In *European Conference on Artificial Intelligence (ECAI)*, pages 63–67, 2006.
6. S. Bistarelli, U. Montanari, and F. Rossi. Semiring-based constraint satisfaction and optimization. *J. ACM*, 44(2):201–236, 1997.
7. S. Bistarelli, U. Montanari, and F. Rossi. Soft concurrent constraint programming. *ACM Trans. Comput. Logic*, 7(3):563–589, 2006.
8. M. G. Buscemi and U. Montanari. CC-Pi: A constraint-based language for specifying service level agreements. In *ESOP'07*, volume 4421 of *LNCS*, pages 18–32. Springer, 2007.
9. P. Codognet and F. Rossi. NMCC programming: Constraint enforcement and retracting in CC programming. In *International Conference on Logic Programming*, pages 417–431, 1995.
10. D. Cohen, M. Cooper, P. Jeavons, and A. Krokhin. The complexity of soft constraint satisfaction. *Artif. Intell.*, 170(11):983–1016, 2006.
11. F. S. de Boer, J. N. Kok, C. Palamidessi, and J. J. M. M. Rutten. Non-monotonic concurrent constraint programming. In *ILPS*, pages 315–334, 1993.
12. R. De Nicola, G. L. Ferrari, U. Montanari, R. Pugliese, and E. Tuosto. A process calculus for qos-aware applications. In *COORDINATION*, volume 3454 of *LNCS*, pages 33–48. Springer, 2005.
13. N. Jennings, P. Faratin, A. Lomuscio, S. Parsons, C. Sierra, and M. Wooldridge. Automated negotiation: prospects, methods and challenges, 2001.
14. A. Keller and H. Ludwig. The WSLA framework: Specifying and monitoring service level agreements for web services. *J. Netw. Syst. Manage.*, 11(1):57–81, 2003.
15. D. D. Lamanna, J. S., and W. Emmerich. SLAng: A language for defining service level agreements. In *Proceedings of FTDCS '03*, page 100. IEEE Computer Society, 2003.
16. V. Saraswat, R. Jagadeesan, and V. Gupta. Default timed concurrent constraint programming. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 272–285. ACM Press, 1995.
17. V. Saraswat and P. Lincoln. Higher-order Linear Concurrent Constraint Programming. Technical report, Xerox Parc, 1992.
18. V. Saraswat and M. Rinard. Concurrent constraint programming. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 232–245. ACM Press, 1990.