A Nonmonotonic Soft Concurrent Constraint Language to Model the Negotiation Process

Stefano Bistarelli
Dipartimento di Matematica e Informatica, Università di Perugia, Via Vanvitelli 1, Perugia, Italy and
Instituto di Informatica e Telematica (CNR), Via Moruzzi 1, Pisa, Italy
bista@dmi.unipg.it and stefano.bistarelli@iit.cnr.it

Francesco Santini
Dipartimento di Matematica e Informatica, Università di Perugia, Via Vanvitelli 1, Perugia, Italy
francesco.santini@dmi.unipg.it

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: retract(c) reduces the current store by c, 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; nask(c) tests if c is not entailed by the store. The new retraction operators also permit to reason about Belief Revision, i.e. the process of changing beliefs to take into account a new piece of information. We present this framework as a possible solution to the negotiation of resources (e.g. web services and network resource allocation) that need a given Quality of Service (QoS). For this reason we also show the the new operators of the language satisfy the Belief Revision postulates [20], which can be used in the negotiation process. The QoS requirements (expressed as semiring levels) of all the parties should converge on a formal agreement through a negotiation process, which specifies the contract that must be enforced.

*Research partially supported by MIUR PRIN 20089M932N project: “Innovative and multi-disciplinary approaches for constraint and preference reasoning”, by CCOS FLOSS project “Software open source per la gestione dell’epigrafia dei corpus di lingue antiche”, and by INDAM GNCS project “Fairness, Equità e Linguaggi”.

Corresponding author
1. Introduction

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 [22] 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 [22]. Notice that this process must be dynamic because clients and providers can change (restrict or relax) 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) [9] 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 [8]. 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. Regarding 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.

Moreover, we show how we can use the new defined retraction operators (i.e. retract and update) to model retraction and contraction operators in Belief Revision [20]: we prove that the nmsccp operators can satisfy AGM postulates [1]. The logical formalization of Belief Revision, which is the process of changing beliefs to take into account a new piece of information, is
investigated in philosophy, in databases, and in artificial intelligence for the design of rational agents [20]. Obviously, to change the current belief some nonmonotonic operators are needed. Notice that Belief Revision and the negotiation have already been studied in conjunction in [32], where negotiation process is seen as a course or multiple courses of mutual belief revision (see Sec. 4).

We apply these extensions to model Service Level Agreements (SLAs) [5, 23, 21] 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).

Other possible motivations for a nonmonotonic constraint-based language [15] are represented by an interactive graphic system, or by the process of debugging a program, where the interaction between the executions of the program and the user may need the deletion and/or the addition of constraints. In [14] the motivations are the same as for this work: (soft) constraints applied to manage SLAs with QoS objectives. In that work, both the provider and the client add their (money) cost constraints to the store and negotiate for the units of purchased bandwidth related to a Web Hosting Service [14]. The main benefit from using our language w.r.t. to other proposals (crisp or “soft”, as [14]) is that at the end of the negotiation we can choose the best solution if there are many suitable possibilities to sign different contracts: the + operator of the semiring defines an ordering among the preferences related to the final solutions (see Sec. 2 and Sec. 6).

The paper extends the idea presented in [10] by expanding the concepts and examples of the original paper and adding all the considerations about Belief Revision (i.e. Sec. 4). 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. Section 4 shows that the retract and update operators of the language satisfy the AGM postulates for retraction and contraction. In Sec. 5 we show how the language can be used to represent preference-driven negotiations. At last, Sec. 6 shows related works and Sec. 7 concludes by indicating future research directions.

2. Background

Absorptive Semirings. An absorptive semiring [7] $S$ can be represented as a $(A, +, \times, 0, 1)$ tuple such that: i) $A$ is a set and $0, 1 \in A$; ii) $+$ is commutative, associative and $0$ is its unit element; iii) $\times$ is associative, distributes over $+$, $1$ is its unit element and $0$ is its absorbing
element. Moreover, $+$ is idempotent, $1$ is its absorbing element and $\times$ is commutative. Let us consider the relation $\leq_S$ over $A$ such that $a \leq_S b$ if $a + b = b$. Then it is possible to prove that (see [8]): (i) $\leq_S$ is a partial order; (ii) $+$ and $\times$ are monotonic on $\leq_S$; (iii) $0$ is its minimum and $1$ its maximum; (iv) $(\mathcal{A}, \leq_S)$ is a complete lattice and, for all $a, b \in A, a + b = \text{lub}(a, b)$ (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 [7] 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 $b \times c = a$ [7]. Absorptive and invertible semirings are 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 $a \leq b$ [7]. Moreover, an absorptive semiring 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 $a \div b$. 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 [7] it is stated that if we have an absorptive and complete semiring, 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 $\div$ operator 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).

In this paper we will consider only invertible semirings, in order to be able to apply $\div$ and to consequently remove information (see the $\oplus$ operator definition in the next Paragraph).

**Soft Constraint Systems.** A soft constraint [8] may be seen as a constraint where each instantiation of its variables has an associated preference. Given $S = (A, +, \times, 0, 1)$ 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 $C = \eta : A \times V \to A$ is the set of all possible constraints that can be built starting from $S, D$ and $V$.

Any function in $C$ 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). The function $\text{supp}(c)$ returns all the variables that belong to the support/scope of $c$. Note that $c\eta[v := d_1]$ means $c\eta'$ where $\eta'$ is $\eta$ modified with the assignment $v := d_1$ (for all $\eta$ assignments). Notice also that, with $c\eta$, the result we obtain is a semiring value, i.e. $c\eta = a$.

Given the set $C$ and for all $\eta$ assignments, the combination function $\ominus : C \times C \to C$ is defined as $(c_1 \ominus c_2)\eta = c_1\eta \times c_2\eta$ (see also [8, 9]); considering the support, $\text{supp}(c_1 \ominus c_2) = \text{supp}(c_1) \cup \text{supp}(c_2)$. Having defined the operation $\div$ on semirings, the constraint division function $\oslash : C \times C \to C$ is instead defined as $(c_1 \oslash c_2)\eta = c_1\eta \div c_2\eta$ [7].

---

1If $S$ 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.
The condition to apply the $\otimes$ operator on $c_1 \otimes c_2$ is that $supp(c_2) \subseteq supp(c_1)$. In fact, since $\otimes$ is the inverse operation of $\otimes$, and since $supp(c_1 \otimes c_2) = supp(c_1) \cup supp(c_2)$, then $supp(c_1) = supp(c_2) \cup supp(c_1 \otimes c_2)$.

Informally, performing the $\otimes$ or the $\otimes$ 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. As they are defined, the $\otimes$ and $\otimes$ operators respectively inherit the properties of $\times$ and $\otimes$. The partial order $\leq_S$ over $C$ can be easily extended among constraints by defining $c_1 \subseteq c_2 \iff \epsilon_1 \leq \epsilon_2$ (for all $\eta$ assignments).

Consider the set $C$ and the partial order $\subseteq$. The entailment relation $\vdash C \subseteq C \times C$, which we will intensively use in Sec. 3 and Sec. 4, is defined s.t. for each $C \in \phi(C)$ and $c \in C$, we have $C \vdash c$ if and only if $\forall C \subseteq c$ and $supp(c) \subseteq supp(C)$. Notice that the condition on the support, that is $supp(c) \subseteq supp(C)$, is new with respect to the works in [9]: in this paper we extend the definition in [9] by requiring that the support of the logical consequence (i.e. $c$) is a subset of $C$. This because it is natural to imply constraints only if the they are defined on the same variables of the premise: in fact, a (set of) constraints over a set $X$ of variables can only deduce information on (a subset of) the variables of $X$ (that is, a constraint supported by $x$ and $y$ cannot entail any information over $z$). According to this view, the $\vdash$ operator is used to “build” new information and as an operation of deduction.

Given a constraint $c \in C$ and a variable $v \in V$, the projection [8, 9] of $c$ over $V \setminus \{v\}$, written $c \parallel_{(V \setminus \{v\})}$ is the constraint $c'$ s.t. $c' = \sum_{d \in D} c[\eta[v := d]]$. 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 $x \in X$, the hiding function [9] is defined as $(A_x)\eta = \sum_{d \in D} c[\eta[x := d]]$. Hidden variables cannot be observed by an external observer [3].

To model parameter passing, for each $x, y \in V$ a diagonal constraint [9] is defined as $d_{x,y} \in C$ s.t., $d_{x,y}[x := a, y := b] = 1$ if $a = b$ and $d_{x,y}[x := a, y := b] = 0$ if $a \neq b$. $0$ and $1$ respectively represent the constraints associating $0$ and $1$ to all assignments of domain values; in general, the $a$ function returns the semiring value $a$. Considering a semiring $S = (A, +, 0, 1)$, a domain of the variables $D$, an ordered set of variables $V$ and the corresponding structure $C$, then $SC = (C, \otimes, 0, 1, A_x, d_{x,y})$ is a cylindric constraint system ("a la Saraswat") as shown in Th. 2.1; the proof of the theorem is given in [9]².

**Theorem 2.1. ([9])**

Consider a semiring $S = (A, +, 0, 1)$, a domain of the variables $D$, an ordered set of variables $V$, the corresponding structure $C$ and the class of hiding functions $A_x : C \rightarrow C$. Then $C$ is a cylindric algebra since the following properties are satisfied:

1. $c \vdash A_x c$

²Notice that in scp, algebraic nature is not required, since the algebraic nature of $C$ strictly depends on the properties of the semiring [9].
Figure 1. A soft CSP based on a Weighted semiring.

2. \( c_1 \uplus c_2 \) implies \( \exists \varsigma c_1 \uplus \exists \varsigma c_2 \)

3. \( \exists \varsigma (c_1 \otimes \exists \varsigma c_2) = \exists \varsigma c_1 \otimes \exists \varsigma c_2 \)

4. \( \exists \varsigma \exists \varsigma c = \exists \varsigma \exists \varsigma c \)

A Soft Constraint Satisfaction Problem (SCSP) [8] is defined as \( P = \langle C, \text{con} \rangle \): \( C \) is the set of constraints and \( \text{con} \subseteq V \) is the set of variables of interest for the constraint set \( C \), which however may concern also variables not in \( \text{con} \). The best level of consistency of a SCSP is defined as \( \text{blevel}(P) = \text{Sol}(P) \downarrow_{\emptyset} \), where \( \text{Sol}(P) = (\times C) \downarrow_{\text{con}} \); notice that \( \supp(\text{blevel}(P)) = \emptyset \). We also say that: \( P \) is \( \alpha \)-consistent if \( \text{blevel}(P) = \alpha \); \( P \) is consistent iff there exists \( \alpha > S_0 \) such that \( P \) is \( \alpha \)-consistent; \( P \) is inconsistent if it is not consistent.

An Example. Figure 1 shows a weighted CSP as a graph. For this kind of problems we use the Weighted semiring, defined as \( \langle \mathbb{R}^+, \min, +, +\infty, 0 \rangle \). Variables and constraints are represented respectively by nodes and by undirected arcs (unary for \( c_1 \) and \( c_3 \), and binary for \( c_2 \)), and semiring values are written to the right of each tuple. The variables of interest (that is the set \( \text{con} \)) are represented with a double circle (i.e. variable \( X \)). Here we assume that the domain of the variables contains only elements \( a \) and \( b \). For example, the solution of the weighted CSP of Fig. 1 associates a semiring element to every domain value of variable \( X \). Such an element is obtained by first combining all the constraints together. For instance, for the tuple \( \langle a, a \rangle \) (that is, \( X = Y = a \)), we have to compute the sum of 1 (which is the value assigned to \( X = a \) in constraint \( c_1 \)), 5 (which is the value assigned to \( \langle X = a, Y = a \rangle \) in \( c_2 \)) and 5 (which is the value for \( Y = a \) in \( c_3 \)). Hence, the resulting value for this tuple is 11. We can do the same work for tuple \( \langle a, b \rangle \rightarrow 7, \langle b, a \rangle \rightarrow 16 \) and \( \langle b, b \rangle \rightarrow 16 \). The obtained tuples are then projected over variable \( X \), obtaining the solution \( \langle a \rangle \rightarrow 7 \) and \( \langle b \rangle \rightarrow 16 \). The \( \text{blevel} \) for the example in Fig. 1 is 7 (related to the solution \( X = a, Y = b \)).

3. The Language

The \text{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 \( \sigma \). It is worth to notice that our \text{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 \( c \) (parameter of \text{retract}) can be removed even if it is different from any other constraints.
previously added to \( \sigma \). However, \( c \) can be removed from the store only if it is entailed by the store, that is \( \sigma \vdash c \). The main difference with the same operator in [12, 14, 15] (see also Sec. 6) is that we use soft constraints instead of tokens.

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 \( \text{update}_X(c) \) primitive has been inspired by the work in [12]. It consists of 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 \( \text{support} = X \)). This operation is variable-grained with respect to our \text{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. 5). A different not atomic \text{update} could be defined by combining \text{retract} and \text{tell}. However in some situations we want these operations to be fulfilled together as a single operation: since the language allows for concurrent agents, in some cases it is necessary to have transactional operations to remove and add new information, in order to not leave the store in an “inconsistent” state where a different agent can execute different actions. The \text{update} operator in [12] is implemented with the classical hiding operator of \text{ccp} languages, thus it fully removes all the tokens in which the passed (single) variable is in the scope; our \text{update} takes a set of variables as a parameter instead and the information removal is performed via the projection operator (see Sec. 2), thus some information not directly concerning the variables may be maintained in the store (see the semantics in Fig. 4).

The \( \text{nask}(c) \) operation (crisp examples are in [15, 26]) is enabled only if the current store does not entail \( \neg c \); it is the negative version of \text{ask}, since it detects \text{absence} of information. Note that, in general, \text{ask}(\neg c) \) is different from \text{nask}(c), so it is necessary to introduce a completely new primitive. Consider for example the store \( \{ x \leq 10 \} \): while the action \( \text{nask}(x < 5) \) succeeds, \( \text{ask}(x \geq 5) \) 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 \text{Boolean} semiring (i.e. \( \langle \{0, 1\}, \lor, \land, 0, 1 \rangle \)). It would be difficult to define \( \neg c \) when using \text{Weighted} semirings [8]. A possible definition of negation for semiring structures that are not boolean algebra but residuated lattices is given in Sec. 4. 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. Our \text{nask} has the same behavior as the \text{nask} in [12], except for the fact that we use soft constraints instead of crisp ones.

Given a soft constraint system as defined in Sec. 2 and any related constraint \( c \), the syntax of agents in \text{nmsccp} is given in Fig. 2. \( P \) 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
Figure 2. Syntax of the nmsccp language.

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 $\rightsquigarrow$ in the syntax notation, which can be considered as a general “checked” transition of the type $\rightsquigarrow^{q_i}_{a_i}$ (e.g., referring to Fig. 2, we can write $\textit{ask}(c) \rightsquigarrow^{q_i}_{a_i} A$, where $q_i$ is a placeholder that can stand for either a semiring element $a_i$ or a constraint $\phi_i$, i.e. $q_i ::= a_i | \phi_i$.

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. $\phi_i$), we need to make a pointwise comparison between the store and the $\phi_i$ constraint (with the $\sqsubseteq$ operator). The way we compare these values/constraints depends on their level in the transition symbol: $a_1$ (or $\phi_1$) 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 $\phi_2$) to prune computations that are too good (i.e. an upper bound). The four possible instantiation of $\rightsquigarrow$ are given in Fig. 3, i.e. $\rightsquigarrow^{a_2}_{a_1}, \rightsquigarrow^{a_2}_{\phi_1}, \rightsquigarrow^{a_2}_{\phi_1}$ and $\rightsquigarrow^{\phi_2}_{\phi_1}$ (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 $\phi_1$ and $\phi_2$ 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 $\textit{ask}$ and $\textit{tell}$ operations in sccp (where only the lower bound is present) can be obtained also in nmsccp: e.g. $\textit{ask}/\textit{tell}(c) \rightarrow^{\alpha}_{\phi_1} A$ and $\textit{ask}/\textit{tell}(c) \rightarrow^{\alpha}_{a_1} A$ respectively represent their sccp not valued and valued versions. The valued rules check for the $\alpha$-consistency of the SCSP, while the not valued ones are a finer check of the store: in this case, a pointwise comparison between the store and the constraint $c$ is performed (see Fig. 3 for the semantics).
3.1. The Operational Semantics

To give an operational semantics to our language we need to describe an appropriate transition system \((\Gamma, T, \rightarrow)\), where \(\Gamma\) is a set of possible configurations, \(T \subseteq \Gamma\) is the set of terminal configurations and \(\rightarrow \subseteq \Gamma \times \Gamma\) is a binary relation between configurations. The set of configurations is \(\Gamma = \{(A, \sigma)\}\), where \(\sigma \in C\) while the set of terminal configurations is instead \(T = \{(success, \sigma)\}\). The transition rules for the nmsccp language are defined in Fig. 4.

The \(\rightarrow\) is a generic checked transition used by several actions of the language. Therefore, to simplify the rules in Fig. 4 we define a function \(\text{check}_{\rightarrow} : \sigma \rightarrow \{\text{true}, \text{false}\}\) (where \(\sigma \in C\), that, parameterized with one of the four possible instances of \(\rightarrow\) (C1-C4 in Fig. 3), returns true if the conditions defined by the specific instance of \(\rightarrow\) are satisfied, or false otherwise. The conditions between parentheses in Fig. 3 claim that the lower threshold of the interval clearly cannot be “better” than the upper one, otherwise the condition is intrinsically wrong.

\[
\begin{align*}
\text{C1: } & \quad \sigma \models \phi_1 \quad \text{iff} \\
& \quad \begin{cases}
\sigma \not\models S a_1 \quad \sigma \not\models S a_2 \\
\sigma \not\models S a_1 
\end{cases} \\
& \quad (\text{with } a_1 \not\prec a_2) \\
\text{C2: } & \quad \sigma \models \phi_2 \quad \text{iff} \\
& \quad \begin{cases}
\sigma \not\models S a_1 \quad \sigma \not\models S a_2 \\
\sigma \not\models S a_1 
\end{cases} \\
& \quad (\text{with } a_1 \not\prec a_2, \not \models a) \\
\text{C3: } & \quad \sigma \models \phi_2 \quad \text{iff} \\
& \quad \begin{cases}
\sigma \not\models S a_2 \\
\sigma \not\models S a_1 
\end{cases} \\
& \quad (\text{with } \phi_1 \models \phi_1) \\
\text{C4: } & \quad \sigma \models \phi_2 \quad \text{iff} \\
& \quad \begin{cases}
\sigma \not\models S a_2 \\
\sigma \not\models S a_1 
\end{cases} \\
& \quad (\text{with } \phi_1 \not\models \phi_1)
\end{align*}
\]

Otherwise, within the same conditions in parentheses, \(\text{check}_{\rightarrow} = \text{false}\)

Figure 3. Definition of the check function for each of the four checked transitions.

Notice that in Fig. 3 we use \(\not\models S a_1\) instead of \(\geq S a_1\) because we can possibly deal with partial orders. Similar considerations can be done for \(\not\models\) instead of \(\models\).

Some of the intervals in Fig. 3 (C1, C2 and C3) are checked by considering the \(a\)-consistency (see Sec. 2) of the \(\sigma\) store: for example, in Fig. 3 C1 checks if the \(a\)-consistency of \(\sigma\) 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.

We provide a description of the transition rules in Fig. 4. In the \textbf{Tell} rule (R1), if the store \(\sigma \otimes c\) satisfies the conditions of the specific \(\Rightarrow\) transition of Fig. 3, then the agent evolves to the new agent \(A\) over the store \(\sigma \otimes c\). Therefore the constraint \(c\) is added to the store \(\sigma\). The conditions are checked on the (possible) next-step store: i.e. \(\text{check}(\sigma')_{\rightarrow}\).

To apply the \textbf{Ask} rule (R2), we need to check if the current store \(\sigma\) entails the constraint \(c\) and also if the current store is consistent with respect to the lower and upper thresholds defined by the specific \(\Rightarrow\) transition arrow: i.e. if \(\text{check}(\sigma)_{\rightarrow}\) is true.
Parallelism and nondeterminism: the composition operators $+$ (represented in R5 with $\Sigma$ when applied to the $E$ agents described in Fig. 2, i.e. ask and nask) and $||$ respectively model nondeterminism and parallelism. A parallel agent (rules R3 and R4) will succeed when both agents succeed. This operator is modeled in terms of interleaving (as in the classical ccp): each time, the agent $A || B$ 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. $\sigma \not\vdash c$.

Retract: with R7 we are able to “remove” the constraint $c$ from the store $\sigma$, using the $\ominus$ constraint division function defined in Sec. 2. The conditions are that it is possible to remove the constraint from the store, that is $\sigma$ entails $c$ (i.e. $\sigma \vdash c$), and the consistency interval satisfies the new store $\sigma \ominus c$.

For example, consider the $c_1$, $c_2$ and $c_4$ weighted constraints in Fig. 5: the domain of the variable $x$ is $\mathbb{N}$ and the adopted semiring is instead the classical Weighted semiring $(\mathbb{R}, \min, +, +\infty, 0)$. It is possible to remove $c_1$ from $c_2$ (because $c_2 \vdash c_1$), i.e. $c_2 \ominus c_1$, and the result is represented by $c_4$ in Fig. 5.

Clearly, it is also possible to completely remove a constraint as if using tokens:

**Theorem 3.1.** (Complete removal) Given a soft constraint system $C$ with an non-idempotent $\times$, where the semiring $S$ is invertible by residuation and thus $\ominus$ can be defined, then the nmsccp agent $\langle \text{tell}(c) \rangle \rightarrow \text{retract}(c) \rightarrow A, \sigma_k$ is equivalent (i.e. the final store is the same) to $\langle A, \sigma_k \rangle$, for every constraint $c_i$, store $\sigma_k$ and $\Rightarrow$ (if
c_1 : (\{x\} \rightarrow \mathbb{N}) \rightarrow \mathbb{R}^+ \quad \text{s.t.} \quad c_1(x) = x + 3 \quad \text{c}_2 : (\{x\} \rightarrow \mathbb{N}) \rightarrow \mathbb{R}^+ \quad \text{s.t.} \quad c_2(x) = 2x + 8
\]
\[
c_3 : (\{x\} \rightarrow \mathbb{N}) \rightarrow \mathbb{R}^+ \quad \text{s.t.} \quad c_3(x) = 2x \quad \text{c}_4 : (\{x\} \rightarrow \mathbb{N}) \rightarrow \mathbb{R}^+ \quad \text{s.t.} \quad c_4(x) = x + 5
\]
\[
c_5 : (\{x\} \rightarrow \mathbb{N}) \rightarrow \mathbb{R}^+ \quad \text{s.t.} \quad c_5(x) = 3 \quad \text{c}_6 : (\{y\} \rightarrow \mathbb{N}) \rightarrow \mathbb{R}^+ \quad \text{s.t.} \quad c_6(y) = y + 1
\]

Figure 5. Six weighted soft constraints (notice that \(c_2 = c_1 \otimes c_4\)). 3 in \(c_5\) is the function that returns the semiring value 3 for all the assignments of \(x\) (see Sec. 2).

enabled). The condition \(c_i \leftarrow c_i\) is clearly always satisfied.

**Proof:**
The agents’ equivalence comes from the properties explained in [2], i.e. \(a \times b \div b = a\) always holds (when \(\times\) is not idempotent), given any two elements \(a, b \in S\). Since the constraint operations (\(\otimes\) and \(\oslash\)) are derived from their related semiring operators (\(\times\) and \(\div\)), the same properties hold.

The semantics of **Update** rule (R8) [12] 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 \(V\setminus X\), 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. \(check(o')\). 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 is based on the projection operator (i.e. \(\|\) in Sec. 2) while the retract is based on the constraint division operator (i.e. \(\oslash\) in Sec. 2). 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 “relaxes” one (or more) variable \(x\) by choosing the best semiring value for each constraint \(c\) supported by \(x\) (i.e. \(a \|_{(V\setminus\{x\})} = \sum_{d \in D} c\eta[x := d]\), 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 \(\langle tell(c_5) \rightarrow_0 retract(c_5) \rightarrow_0 tell(c_2), \bar{0}\rangle\) (in the Weighted semiring \(\bar{1} \equiv \bar{0}\)) results in the store \(c_5 \otimes c_5 \otimes c_2 = c_2\) (since \(c_5 \oslash c_5\) satisfies R7), while \(\langle tell(c_5) \rightarrow_0 update_{\{x\}}(c_2), \bar{0}\rangle\) results in the store \(c_3 \otimes c_2\) (i.e. \(c_5 \otimes c_2\)), since \(\bar{3} = c_5 \|_{(V\setminus\{x\})}\) (see Fig. 5).

**Hidden variables:** the semantics of the existential quantifier in R9 is similar to that described in [28] by using the notion of freshmess of the new variable added to the store (i.e. the variable has not occurred before); freshness is a well-known property in process algebra [3]. Hidden variables cannot be observed by an external observer.

**Procedure calls:** the semantics of the procedure call (R10) (i.e. the call to a different portion of code) has already been defined in [9]: the notion of diagonal constraints (as defined in Sec. 2)
is used to model parameter passing.

To exemplify the rules in Fig. 4, in Ex. 3.1 we show an example where we evaluate an $A \parallel B$ nmsccp agent in the starting empty store $1$ (i.e. the store with empty support). All the checked transitions of the agent (i.e. of the tell, ask and retract actions) are of the type $C1$ in Fig. 3: as a reminder, the $\rightarrow$ transition arrow is a left associative operator, that is nmsccp operations are executed from left to right within the definition of the same agent. The weighted constraints $c_1$ and $c_4$ of the example are represented in Fig. 5, therefore the level of consistency is modeled with the Weighted semiring $(\mathbb{R}^+, \min, +, +\infty, 0)$.

Example 3.1.

\[ \langle (\text{tell}(c_1 \otimes c_4) \rightarrow_7 \text{ask}(c_1) \rightarrow_7 \text{success}) \parallel (\text{retract}(c_4) \rightarrow_7 \text{success}), I \rangle \]

According to $R3$ in Fig. 4 about interleaving parallelism, tell is executed because the retract is not enabled at the first step: according to $R7$, $c_4$ can be removed only if entailed by the store, but the store is empty at the beginning of the computation. Action tell is enabled ($R1$ and $C1$) because $7 \preceq (1 \otimes c_1 \otimes c_4) \not\preceq +\infty$ is satisfied. We have that $(c_1 \otimes c_4) \eta = 8$ (when $x = 0$): following the definition of constraint combination given in Sec. 2, $(c_1 \otimes c_4) \eta = c_1 \eta \times c_4 \eta = (x+3)+(x+5) = 2x+8$. Thus, $A \parallel B$ moves to the state

\[ \langle (\text{ask}(c_1) \rightarrow_7 \text{success}) \parallel (\text{retract}(c_4) \rightarrow_7 \text{success}), c_1 \otimes c_4 \rangle \]

Agent $A$ now suspends on the ask, since, according to $R2$ and $C1$, to be enabled it should be $c_1 \otimes c_4 \vdash c_1$ (and this is true since $c_1 \otimes c_4 \vdash c_1$) and $2 \preceq (c_1 \otimes c_4) \not\preceq 7$. Unfortunately, the previous tell has brought the $\alpha$-consistency (i.e. $\sigma \not\preceq 0$) to 8, which is worse than the lower threshold requested by the ask transition (i.e. $8 \not\preceq 7$ is false). In classical scp, the agent $A$ would consequently fail, but in this case the retract of the agent $B$ can increase the consistency of the store. The retract is enabled if ($R7$ and $C1$) $\sigma \vdash c_4$ and $2 \preceq (c_1 \otimes c_4 \otimes c_4) \not\preceq +\infty$. Since $\sigma = c_1 \otimes c_4 \vdash c_4$, the $\alpha$-consistency is 8 and $c_1 \otimes c_4 \otimes c_4 = c_1$, the new state is

\[ \langle (\text{ask}(c_1) \rightarrow_7 \text{success}) \parallel \text{success}, c_1 \rangle \]

Now that $c_1 \vdash c_1$ and $c_1 \not\preceq 3$, the ask can be finally performed because the conditions given by $R2$ and $C1$ are now satisfied:

\[ \text{success} \parallel \text{success}, c_1 \rightarrow \text{success}, c_1 \]

Similar considerations can be formulated for other actions of the language in Fig. 4: an action can be suspended, but has always a chance to be re-enabled. Notice that in nmsccp a tell can enable another suspended tell because of the upper threshold of the consistency interval: suppose the following agent $A \parallel B$:

\[ \langle (\text{tell}(c_1) \rightarrow_7 \text{success}) \parallel (\text{tell}(c_4) \rightarrow_7 \text{success}), I \rangle \]

Because $(I \otimes c_1) \not\preceq 3$, the tell of $A$ cannot be enabled, according to $R1$ and $C1$, because it should be $7 \preceq (I \otimes c_1) \not\preceq +\infty$, and $7 \not\preceq 3$ is false. But the tell of $B$ is instead enabled and executed, since $(I \otimes c_4) \not\preceq 5$, which is worse than the upper threshold, i.e. 4. Now, since after
the tell of B the current store is changed (i.e. \( \sigma = c_4 \)), and \( (c_1 \otimes c_4) \sqcap \emptyset = 8 \), the tell of agent A is finally enabled because the checks of the transition are now satisfied. To sum up, this language has no failure agent in Fig. 2 and we do not need transition rules to capture agents’ failure.

### 3.2. Preference Representation and Operations

The representational and computational issues are complex [16]. Some 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 \( c_7(x, y) = x^2 - 3x + 4y \), \( c_8(x) = 3x + 2 \) and \( c_0(y) = 3y + 2 \): if the initial store contains \( c_7(x, y) \), tell\( (c_8) \) gives \( (c_7 \otimes c_8) = x^2 - 3x + 4y + 3x + 2 = x^2 + 4y + 2 \), and then a retract\( (c_8) \) (possible because \( \sigma \vdash c_0 \) according to R7 in Fig. 4) would result in the store preference \( (c_7 \otimes c_8 \otimes c_0) = x^2 + 4y + 2 - (3y + 2) = x^2 + y \). To compute the result of an update\( [y] (c_4) \) we need to project over \( V \setminus \{y\} \) (see Sec. 2) before adding \( c_{10} \); therefore, if the store preference is \( x^2 + y \), we must find the minimum of this polynomial varying \( y \), and this minimization gives \( y = 0 \), finally obtaining \( x^2 \otimes c_4 = x^2 + x + 5 \) as result (see Fig. 5). 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 example, if the agents have chronologically performed the actions tell\( (c_7) \), tell\( (c_8) \), retract\( (c_9) \) and update\( X (c_4) \), the store will be \( c_7 \otimes c_8 \otimes c_9 \sqcup_{\{V\setminus X\}} (\otimes c_4 \) (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 ordering of the actions is important.

Given a soft constraint system \( C \), where the semiring \( S \) is invertible by residuation, changing the tell and retract actions ordering inside an agent may change the final store, as described in the following two examples.

**Example 3.2.** If we suppose the \( \times \) of \( S \) as idempotent, we have the nmscep agent \( \langle \text{tell}(c_i) \mapsto \text{retract}(c_i) \mapsto \text{tell}(c_i) \mapsto A, \sigma_k \rangle \equiv \langle A, c_i \otimes \sigma_k \rangle \), and by changing the ordering of actions it differs from \( \langle \text{tell}(c_i) \mapsto \text{tell}(c_i) \mapsto \text{retract}(c_i) \mapsto A, \sigma_k \rangle \equiv \langle A, \sigma_k \rangle \), for every constraint \( c_i \), store \( \sigma_k \) and \( \mapsto \) (if enabled). To prove it, we consider that for every semiring element \( a \in S \), we have \( (a \times a) \div a = 1 \) (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 [7].

**Example 3.3.** If we instead suppose to have a not idempotent \( \times \), we have that (see the constraints in Fig. 5) \( \langle \text{tell}(c_2) \mapsto \text{retract}(c_4) \mapsto \text{success}, c_1 \rangle \) successfully terminates with the store \( c_1 \otimes c_2 \otimes c_4 = 2x + 6 \) (since \( c_1 \otimes c_2 + c_4 \) according to R7 in Fig. 4), while \( \langle \text{retract}(c_4) \mapsto \text{tell}(c_2) \mapsto \text{success}, c_1 \rangle \) is blocked on the retract since \( \sigma = c_1 \not\vdash c_4 \) (according to R7 in Fig. 4).
This representation (i.e. keeping also the sequence of operations) differs from the classical one given by Saraswat [28] or in [14], since in these works a retract removes from the store only one instance of the token: \( \langle \text{tell}(c_1) \rightarrow \text{tell}(c_1) \rightarrow \text{retract}(c_1) \rightarrow A, \bar{1} \rangle \equiv \langle A, c_1 \rangle \), even if \( \times \) is idempotent. Therefore, the ordering of the actions is unimportant and the store can be seen only as a set of tokens.

4. NMSCCP and AGM Postulates

The AGM postulates [1] (named “AGM” after the initials of the authors Alchourrón, Gärdenfors, and Makinson) formalize the field of belief change with properties that an operator should satisfy in order to obtain a rational agent. Belief Revision [20] is the process of changing beliefs to take into account a new piece of information: this process is non-trivial since several different ways for performing this operations may be possible and AGM postulates are a well-known and accepted base in the community. These needs mainly arise in multi-agent communities, where different information comes from different sources. The intuition upon which the AGM postulates were based is independent of the peculiarities of the underlying logic. Three operations were originally considered: expansion (addition of a belief without a consistency check), revision (addition of a belief while maintaining consistency), and contraction (removal of a belief). Other operators usually defined in Belief Revision consist in consolidation (restoring consistency of a set of beliefs) and merging (fusion of two or more sets of beliefs while maintaining consistency). The authors of [1] introduced a set of postulates for contraction and revision that formally describe the properties that such operators should satisfy.

A related philosophical consideration studies how the knowledge should be represented [19]. There are two viewpoints: foundational theories and coherence theories [19]. Under the foundational viewpoint, each piece of knowledge serves as a justification for other beliefs; this viewpoint is closer to the belief base approach. On the other hand, under the coherence model, no justification is required for the beliefs; a belief is justified by how well it fits with the rest of beliefs, in forming a coherent and mutually supporting set; thus, every piece of knowledge helps directly or indirectly to support any other. Coherence theories match with the use of belief sets as a proper knowledge representation format. The classical AGM theory is based on the coherence model (using belief sets) [19].

The goal of this section is to show that the operators provided with the nmscc language are AGM-compliant. The application of Belief Revision to negotiation will be described at the end of this section. In the following, the \( \ominus \) symbol stands for the revision operator, the \( \odot \) symbolizes the general expansion operator and \( \otimes \) is the contraction operator. In our case, the Knowledge Base (or KB i.e. the set of all the pieces of knowledge available to the reasoning agents) is represented by the constraint store (i.e. \( \sigma_K \)), to and from which a soft constraint \( c_\alpha \), which represents the piece of belief \( \alpha \), can be added or removed. The three main operators revision, expansion and contraction are for us represented by tell, update and retract respectively (see Sec. 3.1 for their semantics): \( K \odot \alpha \equiv \langle \text{update}(c_\alpha)_{|V \setminus \text{supp}(c_\alpha)} \rightarrow 1 A, \sigma_K \rangle \), \( K \ominus \alpha \equiv \langle \text{tell}(c_\alpha) \rightarrow 1 A, \sigma_K \rangle \) and \( K \otimes \alpha \equiv \langle \text{retract}(c_\alpha) \rightarrow 1 A, \sigma_K \rangle \). As a remind, the support of \( c_\alpha \) (i.e. \( \text{supp}(c_\alpha) \)) is the set of variables on whose assignment the soft constraint depends on, and \( V \) is the whole set of the problem variables (see Sec. 2).
In the following we prove that the “AGM basic postulates for contraction” A1-A6, concerning the contraction operator (i.e. $\Box$) can be satisfied in our language.

In the A1-A6 postulates, the consequence operator, denoted by $Cn$, allows us to conclude facts from other facts; the implications $Cn(X)$ of a belief $X$ are determined by the entailment operator, that is, by using soft constraints, $Cn(c_X)$ is a set $Y$ of soft constraints s.t. $\forall c_i \in Y, c_X \vdash c_i$, i.e. $Cn(c_X)$ represents the set of all constraints implied by constraint $c_X$. Therefore, we directly adopt the definition $X \vdash Y \iff Y \subseteq Cn(X)$, already given in [19]. Both the $\exists$ and $\forall$ operators are naturally interpreted as $\vdash$ (see Sec. 2): $\alpha \in K$ and $\alpha \subseteq K$ are interpreted as $\sigma_K \vdash c_{\alpha}$.

Moreover, notice that the A4 postulate only holds for the Boolean semiring; this is not a limitation of our semiring-based framework, but an issue that involves other weighted approaches to belief revision [13], where not all the postulates always hold: for the frameworks where $\alpha \cap (K - \alpha) \neq \emptyset$, e.g. in fuzzy theory or for multisets, the A4 postulate does not hold.

- A1 $\equiv K \otimes \alpha$ is a belief set: when we remove a constraint $c_{\alpha}$ from the store $\sigma_K$ we obtain a new store $\sigma'_K = \sigma_K \ominus c_{\alpha}$, which represents the new belief set.

- A2 $\equiv K \otimes \alpha \subseteq K$: in constraints we translate it as $\sigma_K \vdash \sigma_K \ominus c_{\alpha}$. This postulate is true according to the properties of $\ominus$ (see Sec. 2), that is, $\times$ (and consequently $\ominus$) is a monotonic operator over $\leq_S$.

- A3 $\equiv$ if $\alpha \notin Cn(K)$, then $K \otimes \alpha = K$. Since contraction involves the retract operation, we refer to rule R7 in Fig. 4, where it is stated that the retract can be executed only when $\sigma_K \vdash c_{\alpha}$. Since $\sigma_K \not\vdash c_{\alpha}$ (according to the premise of the postulate), this implication is always true.

- A4 $\equiv$ if $\alpha \notin Cn(\emptyset)$, then $\alpha \notin Cn(K \otimes \alpha)$. This postulate holds for the Boolean semiring\footnote{When using a different semiring (i.e. not the Boolean semiring), it is always possible that if we remove a constraint from the store, the same constraint is still implied by it, e.g. $(c_a \otimes c_{\alpha}) \ominus c_a \vdash c_{\alpha}$ with $c_{\alpha} = c_a \otimes c_{\alpha}$ in the Weighted semiring, or $\sigma_K \ominus c_{\alpha} \vdash c_{\alpha}$ in the Fuzzy semiring (i.e. $\{0, 1\}, \max, \min, 0, 1$) when $c_{\alpha} \subseteq c_a$, since $\sigma_K \ominus c_{\alpha} = \sigma_K$ (see Sec. 2). This fact concerns all the weighted approaches to belief revision [13], where the A4 postulate does not hold.}: when $\bar{1} \not\vdash c_{\alpha}$, that is the premise $\alpha \not\subseteq Cn(\emptyset)$ (where $\bar{1}$ is the constraint that assigns true to each variable assignment), then the consequence can be translated in constraints as $\sigma_K \ominus c_\alpha \not\vdash c_{\alpha}$. In words, by removing some information from the store, that is $c_{\alpha}$, (i.e. $\bar{1} \not\vdash c_{\alpha}$), the store cannot entail it anymore (i.e. $\sigma_K \ominus c_{\alpha} \not\vdash c_{\alpha}$), as in the crisp constraint systems based on tokens [14, 30, 15, 26].

- A5 $\equiv$ if $Cn(\{\alpha\}) = Cn(\{\beta\})$, then $K \otimes \alpha = K \otimes \beta$: if $Cn(\{\alpha\}) = Cn(\{\beta\})$ then $c_{\alpha} = c_{\beta}$, and $\sigma_K \ominus c_{\alpha} = \sigma_K \ominus c_{\beta}$ simply follows from the monotonic properties of $\ominus$ (see Sec. 2).

- A6 $\equiv$ if $K \subseteq Cn((K \otimes \alpha) \cup \{\alpha\})$: this postulate is true since $(\sigma_K \ominus c_{\alpha}) \ominus c_{\alpha} = \sigma'_K$ and $\sigma'_K \vdash \sigma_K$. When $\sigma_K \vdash c_{\alpha}$ (i.e. when the retract can be executed), then $(\sigma_K \ominus c_{\alpha}) \ominus c_{\alpha} = \sigma_K$.

To prove some of the following postulates for the revision operator (i.e. in B4 and B8) we need the definition of preference negation:
Definition 4.1. (Residuated negation)
Given a semiring \( (A, +, \times, 0, 1) \) and \( a, b \in A \), we define the residuated [11] negation of \( a \) as \( \neg a = \max \{ b : b \times a = 0 \} \), where \( \max \) is according to the ordering defined by \( + \).

Notice that over the boolean semiring the negation operator exactly corresponds to the logic negation, since \( \neg 0 = \max \{ b : b \times 0 = 0 \} \) and \( b = 1 \), while when \( \neg 1 = \max \{ b : b \times 1 = 0 \} \) then the only possibility is \( b = 0 \). When the considered semiring has no 0 divisors (i.e. \( a \times b = 0 \) only when \( a = 0 \) or \( b = 0 \)), then \( \neg a = 0 \) (for every \( a \neq 0 \)). The negation can be clearly extended from semiring values to soft constraints:

Definition 4.2. (Soft constraint negation)
Given a soft constraint \( c = (V \to D) \to A \) and a semiring \( (A, +, \times, 0, 1) \) with \( a \in A \), \( \neg c \) is a constraint such that if \( c\eta = a \) then \( \neg c\eta = \neg a \), where \( \neg a \) is defined as in Def. 4.1.

As a sound consequence, \((c \otimes \neg c)\eta = \emptyset\) represents \( p \land \neg p = \text{false} \) in boolean algebra. Notice also that \( \text{supp}(c) = \text{supp}(\neg c) \). The constraint negation in Def. 4.2 is needed to deal with the operator of contraction, which is related with the operator of revision by the Levi identity (Eq. Levi Equation):

\[
K \odot P = (K \odot \neg P) \odot P
\]

(Levi Equation)

Whenever an operator satisfies the six postulates for contraction, its corresponding revision operator satisfies the eight postulates for revision, and vice versa [19]. Therefore, we can define our \( \odot \) revision operator as

\[
K \odot P = \sigma_K \otimes \neg c_P \otimes c_P
\]

Since the proofs of A1-A6 where the contraction is performed with the \( \oplus \) operator hold, the \( \odot \) revision operator as defined above (i.e. remove \( \neg c_P \) and then add \( c_P \)) automatically satisfy the postulates of revision according to the Levi identity. However, we want to show that also the update operator used in the nmscpcp language satisfy the postulates of revision. For each of the following postulates B1-B8, we first state the original definition, then we show how they are implemented in our language and prove that the operators we introduced satisfy the postulates. We prove them by directly using the semantics of the operations provided in Fig. 4, e.g. \( \text{update}(c_a)_{V \setminus \text{supp}(c_a)} \) in the store \( \sigma_K \) is equivalent to \( \sigma_K \downarrow_{V \setminus \text{supp}(c_a)} \otimes c_a \). The first six postulates (i.e. B1-B6) are called “the basic AGM postulates” for revision, while B7 and B9 are called “the supplementary AGM postulates” for revision. The hypotheses needed to prove the following postulates are i) a semiring in which \( a \times b = 0 \implies a = 0 \) or \( b = 0 \) (required by B5), and ii) that when \( \neg a \notin K \), we suppose that \( c_a \) is the only constraint on \( \text{supp}(c_a) \), as supposed before for the contraction postulates (required by B4 and B8).

- B1 \( \equiv K \odot a \) is a belief set. For this postulate we can simply notice that \( \sigma_K \downarrow_{V \setminus \text{supp}(c_a)} \otimes c_a = \sigma'_K \); that is, we obtain an updated store \( \sigma'_K \).

- B2 \( \equiv a \in K \odot a \). This is true also in our language, since \( c_a \ni \sigma_K \downarrow_{V \setminus \text{supp}(c_a)} \otimes c_a \); before adding \( c_a \) the store is relaxed with an update operation (see the definition of \( \downarrow \) in Sec. 2).
• B3 ≡ K ⊖ α ⊆ K ⊕ α. For this postulate we need to prove that σK \{\alpha \} \sqcup c_α \trianglerighteq σK \sqcap c_α, but this is true since σK \{\alpha \} \sqcup c_α \trianglerighteq σK (see the definition of \sqcup in Sec. 2) and \sqcap is monotone.

• B4 ≡ if \neg \alpha \not\subseteq K then K \trianglerighteq \alpha \not\subseteq K \trianglerighteq \alpha. The negation of a constraint is defined in Def. 4.2. We need prove that σK \{\alpha \} \sqcap c_\alpha \trianglerighteq σK \{\alpha \} \sqcup c_\alpha, since one of the two hypotheses is that c_\alpha is the only constraint on supp(c_\alpha) (due to ε and the definition of \sqcap in Sec. 2), we have that σK \{\alpha \} \sqcap c_\alpha = σK, and therefore the postulate is proved.

• B5 ≡ K \triangleleft \alpha is inconsistent ⇐⇒ \alpha is inconsistent. In our case, given σK \neq 0, then σK \{\alpha \} \sqcap c_\alpha = 0 ⇐⇒ c_\alpha = 0. The postulate can be proved by using the first hypothesis, i.e. for all semirings in which \alpha \times b = 0 = \alpha or b = 0.

• B6 ≡ α \leftrightarrow β then K \trianglerighteq α = K \trianglerighteq β. If c_\alpha \trianglerighteq c_\beta then \sigma = \sigma, therefore, the same revision operation performed in the same store σ yields same result.

• B7 ≡ K \trianglerighteq (α \land μ) \not\subseteq (K \trianglerighteq α) \trianglerighteq μ. Naturally, we translate α \land μ to c_\alpha \sqcap c_μ. Therefore, by translating the postulate with soft constraints, we have that σK \{\alpha \} \sqcap (c_\alpha \sqcap c_μ) \trianglerighteq σK \{\alpha \} \sqcup (c_\alpha \sqcap c_μ) is true because \sigmaK \{\alpha \} \sqcup (c_\alpha \sqcap c_μ) = \sigmaK \{\alpha \} \sqcup (c_\alpha \sqcap c_μ).

• B8 ≡ if \neg β \not\subseteq K \trianglerighteq α then (K \trianglerighteq α) \trianglerighteq β \subseteq K \trianglerighteq (α \land μ). We can translate it as (σK \{\alpha \} \sqcup c_\alpha) \sqcap c_μ \trianglerighteq σK \{\alpha \} \sqcup (c_\alpha \sqcap c_μ). Due to the associativity property of \sqcap and by using the hypothesis that c_μ is the only constraint on supp(c_μ) (due to ε and the definition of \sqcap in Sec. 2), the proof is similar to the proof for B4: we have that σK \{\alpha \} \sqcup c_\alpha = σK \{\alpha \} \sqcup (c_\alpha \sqcap c_μ).

At last, we show also the postulates for the third important operator in Belief Revision, i.e. the expansion operator [1] K \trianglerighteq α ≡ (tell(\alpha) \to^{0} A, \sigma), which corresponds to the trivial addition of a sentence to a KB, without taking any special provisions for maintaining consistency. Postulate C4 only holds for idempotent semirings.

• C1 ≡ K’ = K \trianglerighteq α is a belief set. For this postulate we can simply notice that we obtain a new constraint store σ’ = σK \trianglerighteq c_α.

• C2 ≡ α \in K \trianglerighteq α is trivially true since σK \trianglerighteq c_α always holds (see Sec. 2).

• C3 ≡ K \subseteq K \trianglerighteq α is equivalent to σK \trianglerighteq c_α. This always holds because the \times (and consequently \sqcap) is a monotonic operator over ≤S (see Sec. 2).

• C4 ≡ if α \in K then K = K \trianglerighteq α. This postulate can be proved only for semirings where the \times is idempotent, since, if σK \trianglerighteq c_α then σK \trianglerighteq c_α = σK.

• C5 ≡ if H \subseteq K then H \trianglerighteq α \subseteq K \trianglerighteq α. Expressing this with constraints, if σH \trianglerighteq σK, then σH \trianglerighteq c_α \trianglerighteq σK \trianglerighteq c_α. This is true because \times is a monotonic operator (see Sec. 2).

Negotiation is a process of consensus-seeking among two or more agents [22]. If we consider the demands (or offers) of parties as their beliefs on the matter in question, the change of the demands of each party reflects the change of its beliefs during the progress of negotiation [32].
The parties who are convinced to accept part of the other parties demands would perform a belief revision. New belief states of participants represent their revised demands which are normally closer to each other and apt to reach an agreement. In [32], such kind of belief revision is called mutual belief revision and the negotiation functions are assembled in order to satisfy the AGM postulates.

5. An Example: the Negotiation of Service Levels Agreements

An interesting application of the \textit{nmsccp} language is to model generic entities negotiating a formal agreement, i.e. a SLA [5, 23, 21]. The main task consists in accomplishing the requests of all the agents by satisfying their QoS needs. Considering the fuzzy negotiation in Fig. 6 (\textit{Fuzzy} semiring: $\langle [0, 1], \text{max}, \text{min}, 0, 1 \rangle$) both a provider and a client can add their request to the store $\sigma$ (respectively $\text{tell}(c_p)$ and $\text{tell}(c_c)$): the thick line represents the consistency of $\sigma$ after the composition (i.e. \textit{min}), and the \textit{blevel} of this SCSP (see Sec. 3.1) is the \textit{max}, where both requests intersects (i.e. in 0.5). The two constraints $c_p$ and $c_c$ are defined in Fig. 7. In Fig. 6 we use the \textit{Fuzzy} semiring because the goal is to maximize the minimum of the intersection between the two client/provider preferences, in order to not disappoint one of the two parties too much. By having the possibility to use different semirings we can represent different preferences and optimization criteria.

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 \textit{cross-domain} management of services proposed in [5]. Variable $x$ represents the number of failures that can happen during the service provision [21], while the preference of soft constraints models the number of hours (or a money cost in hundreds of euro) needed to manage them and recover from them [21]. The preference interval on transition arrows models the fact that both $P_1$ and $P_2$ may impose a minimum time for the failure recovery procedure (the upper bound in Fig. 3) and a maximum time for the same procedure (lower bound in Fig. 3). In contracts it is usually possible to
define upper and lower time thresholds in which a service must be accomplished: in this case the minimum time represents the necessary time needed to call and wait for the external IT technicians who have to recover from the system failures; the maximum time represents the maximum time within which the service must be operative again, as stated in the contract (i.e. a maximum time for the service of repairing) [21].

For the negotiation of the SLA in this scenario we use the Weighted semiring and the soft constraints given in Fig. 5. In Sec. 3 we use the constraints in Fig. 5 to explain the operational semantics of the language, while in the following of this section we use them to model a negotiation: for example in Ex. 5.1, the preference function represented by $c_4(x) = x + 5$ (in Fig. 5) models the number of hours needed by $P_1$ to manage and recover from a failure, that is 5 hours by default plus one hour for each detected failure. 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.

The four examples (i.e. Ex. 5.1, Ex. 5.2 Ex. 5.3 Ex. 5.4) represent four different steps that two agents ($P_1$ and $P_2$) can perform during their (“life-long”) SLA negotiation process: needs and requests may change during the lifetime of the two companies and thus the negotiation process can be continuous [22]. In Ex. 5.1 the first agreement is set up, while after some time, Ex. 5.2 describes a relaxation of the policy performed by $P_1$. Then, Ex. 5.3 represents an example of nask use to check if some of the requirements asked by $P_2$ are not entailed by the store anymore: this can lead $P_2$ to not sign the agreement, for example. At last, Ex. 5.4 shows how $P_1$ can substantially modify its policy by eliminating some variables (i.e. with ↓ projection in the update operation) which it depends on, in order to base it on new variables.

**Example 5.1. (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: $P_1 = \langle \text{tell}(c_4) \rightarrow_0 \text{tell}(s_{p2} = 1) \rightarrow_0 \text{ask}(s_{p1} = 1) \rightarrow_2 \text{success} \rangle \langle \text{tell}(c_3) \rightarrow_0 \text{tell}(s_{p1} = 1) \rightarrow_0 \text{ask}(s_{p2} = 1) \rightarrow_1 \text{success} \rangle \equiv P_2$, executed in the store with empty support (i.e. 0). Variables $s_{p1}$ and $s_{p2}$ are used only for synchronization and thus will be ignored in the following considerations (they will be replaced by the SYNCHRO$_i$ agents in the following examples for sake of brevity). The two agents synchronize in order to detect the end of the negotiation phase. The final store
(the merge of the two policies) is \( \sigma = (c_4 \otimes c_3) \equiv 2x + x + 5 \), and since \( \sigma \models_0 5 \) 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 5.2. (Retract)

After some time (still considering Ex. 5.1), 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 [6]. 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. 6), is clearly important in a
negotiation process. \( P_1 \equiv \langle \text{tell}(c_4) \rightarrow_0^{\infty} \text{SYNCHRO} P_1 \rightarrow_0^{\infty} \text{retract}(c_1) \rightarrow_0^{\infty} \text{success} \rangle \langle \text{tell}(c_3) \rightarrow_0^{\infty} \text{SYNCHRO} P_2 \rightarrow_0^{\infty} \text{success} \rangle \equiv P_2 \) is executed in \( \bar{0} \). The final store is \( \sigma = c_4 \otimes c_3 \otimes c_1 \equiv 2x + 2 \), and
since \( \sigma \models_0 2 \) and \( (c_4 \otimes c_3) \models_0 c_1 \), both \( P_1 \) and \( P_2 \) now succeed (it is included in both intervals).

Example 5.3. (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. \( P_1 \equiv \langle \text{tell}(c_4) \rightarrow_0^{\infty} \text{SYNCHRO} P_1 \rightarrow_0^{\infty} \text{nask}(c_4) \rightarrow_0^{\infty} \text{tell}(c_4) \rightarrow_0^{\infty} \text{SYNCHRO} P_2 \rightarrow_0^{\infty} \text{success} \rangle \equiv P_2 \) (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 the 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 5.4. (Update)
The update can be instead used for substantial changes of the policy: for example, suppose that
\( P_1 \equiv \langle \text{tell}(c_1) \rightarrow_0^{\infty} \text{update}_{c_4}|c_6 \rangle (c_6) \rightarrow_0^{\infty} \text{success}, \bar{0} \rangle \). This agent succeeds in the store \( \bar{0} \otimes c_1 \downarrow_0\models_1 \otimes c_6 \),
where \( c_1 \downarrow_0\models_1 \otimes 3 = 3 \) and \( 3 \otimes c_6 \equiv y + 4 \) (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 errors in order to deal with
a finer granularity: as a remind, a fault may cause an error, which may cause a failure [24]. 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 \( 3 \) component of the final store derives from the “old” \( c_1 \), meaning that
some fixed management delays are included also in this new policy. For example, in this case it
represents the minimum necessary time needed to call and wait for the external IT technicians
to repair the system: this time is the same even if we now consider errors instead of failures.

Notice that during the negotiation between two parties, e.g. \( P_1 \) and \( P_2 \), \( P_2 \) may relax the
constraints previously added to the store by \( P_1 \) (and viceversa). This is not a security problem,
since \( P_1 \) can always ask if a set of indispensable requirements are still implied by the store after
ending the negotiation process by synchronizing with the other party on this end event (i.e.
with the SYNCHEP protocols in the examples above). Then, it can approve or reject the result of the negotiation. In addition, to solve possible security problems, we can add a label to each constraint describing the “owner” of the added constraint: removing operations may also be labeled with the name of the agent performing the action, which can be prevented if it removes part of a constraint added to the store by a different agent.

6. Related Work

Nonmonotonicity has been extensively studied for crisp constraints in the so-called linear CCP programming [27] and in following works as [4, 15, 12, 26]. The inspiration for this work comes from [15] and [12]: in [12] the authors present a nonmonotonic framework for Concurrent Constraint Programming (CCP) [30], together with its semantics. Our nask and update operations (see Sec. 3) are the soft versions of those described in [12], while the atell (also described in [12]), which adds a constraint only if it is consistent with the store, can be trivially modeled with the classical (valued) tell of CCP. The update operator in [12] is implemented with the classical hiding operator of CCP languages, thus it fully removes all the tokens in which the passed (single) variable is in the scope; our update takes a set of variables as a parameter instead and the information removal is performed via the projection operator (see Sec. 2), thus some information not directly concerning the variables may be maintained in the store (see the semantics in Fig. 4). The nask operator in [12] has the same behavior as the nask in this paper, except for the fact that we use soft constraints instead of crisp ones.

A negative ask like our nask is described also in [29]. The idea for a fine-grained removal of constraints (the retract in Sec. 3) comes from [15], 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 behavior during the negotiation interactions in open systems. Other examples of nonmonotonic evolution of the constraint store in CCP are presented in [18], and their line of research is usually called Linear Concurrent Constraint Programming.

Regarding related SLA negotiation models, the process calculus introduced in [17] 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 [5] and [23] 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 [14], 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. \( \sigma \vdash c \)), even if c is syntactically different from all the constraints previously added (as the retraction of \( c_1 \) in Ex. 5.2). 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 [14], 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 [14]; 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. 3). In this way, each of the agents can specify its desired preference for the final agreement. This is a relevant improvement with regard to [14], where the final store collects all the consistent solutions without any distinction, i.e. each solution that satisfies $\sigma \downarrow_0 = \alpha_i$, for every $\alpha_i > S_0$.

At last, iii) we introduced the update operation (extending the semantics of the crisp update in [12]), which is a variable-grained relaxation, and the nask whose crisp version is in [12], 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 [14] 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. 3 to prevent the store from becoming not consistent “enough”.

Notice that with this different behavior, that is relaxation (with nmsccp) versus complete removal (as in [14]), we can model a different theory in philosophy, i.e. coherentism instead of foundationalism [25, 19]. According to foundationalism approach to Belief Revision, retracting a non-derived piece of knowledge should lead to retracting all its consequences that are not otherwise supported (by other non-derived pieces of knowledge). This approach can be realized by using knowledge bases that are not deductively closed and assuming that all formulae in the knowledge base represent self-standing beliefs, that is, they are not derived beliefs [25, 19]. This approach can be modeled by using a retract operator like the one in [14, 12], which, as a remind, is able to remove only those constraint tokens previously added to the store, and not the constraints implied by them.

In order to distinguish the foundational approach to Belief Revision to that based on deductively closed knowledge bases, the second approach is called the coherentist one. This name has been chosen because the coherentist approach aims at restoring the coherence (consistency) among all beliefs, both self-standing and derived ones [25]. This approach is related to coherentism in philosophy. With our retract, since it is not based on tokens, we can remove also the implied constraints: for example, if we added $x > 5$, we can remove $x > 5$ but also $x > 4$, which is implied by $x > 5$.

Concerning the related work on the AGM postulates, in [13] the authors treat the problem of revising fuzzy belief bases, i.e., belief base revision in which both formulas in the base as well as revision-input formulas can come attached with varying degrees. A general framework for fuzzy logic is able to capture certain types of uncertainty calculi as well as truth-functional fuzzy logics; they show how the idea of rational change from crisp base revision, as embodied by the idea of partial meet (base) revision, can be extended to revising fuzzy belief bases.
7. 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 (task, update and retract) that allow the nonmonotonic evolution of the store. We have chosen to extend sccp because soft constraints [8] 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.

During a negotiation among several parties, if we consider the demands and offers as the beliefs on the matter in question, the change of the demands of each party reflects the change of its beliefs during the progress of negotiation [32]. Therefore, the study of (obviously) non-monotonic languages and the properties of their primitives is very important to understand the negotiation process and what is the current global belief after a change, that is a retraction/addition of information.

We would like to merge this language with the timing mechanisms (e.g. “timeout” and “interrupt”) explained in [6]. 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 [31], 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 applications, and surely for (SLA) negotiating entities, where each agent has a private store collecting its resources (i.e. variables) and policies (i.e. constraints). Finally, we want to study the absence of failures in nmsccp.

References


