A Secure Coordination of Agents with Nonmonotonic Soft Concurrent Constraint Programming*,**

Stefano Bistarelli¹,² and Francesco Santini¹,³

¹ Dipartimento di Matematica e Informatica, Università di Perugia, Italy  
bista,francesco.santini@dmi.unipg.it  
² Istituto di Informatica e Telematica, IIT-CNR, Pisa, Italy  
stefano.bistarelli@iit.cnr.it  
³ Centrum Wiskunde & Informatica, Amsterdam, Netherlands  
F.Santini@cwi.nl

Extended abstract. In the context of distributed/concurrent systems, the ability to coordinate the agents coupled with the possibility to control the actions they perform is significantly important. The necessity of guaranteeing security properties is rapidly arising: in an open and untrusted environment, an attacker can threat the integrity and confidentiality properties of the exposed data. The ingredient at the basis of our research is Nonmonotonic Soft Concurrent Constraint Programming (NSCCP) [4]. The NSCCP language extends the classical Soft Concurrent Constraint Programming (SCCP) language [3] with the possibility of relaxing (i.e. retracting or removing constraints) the store with a retract action, which clearly improves the expressivity of the language [4]. However, non-monotonicity raises further security concerns, since the store σ is a shared and centralized resource accessed in a concurrent manner by multiple agents at the same time: may an agent A relax a constraint c added to σ by the agent B? Since in this case we are reasoning about soft constraints instead of crisp ones, “how much” of c can agent A relax? Even if (S)CCP has been successfully used to analyse security issues [1], the paradox is that security aspects linked to the language itself have not been inspected yet. For these reasons, a constraint-based language modeling the interactions among agents in an untrusted environment needs to support security by providing some access control mechanisms with a granularity at the level of the single constraints. Therefore, our intent is to equip the core actions of the NSCCP language [4] with a formal system of rights on the constraints and then study the execution of agents from this new point of view. We take inspiration from the Access Control List (ACL) model [6], which is one of the security concepts in the design of secure computing systems. An ACL specifies which users or system processes are granted access to objects, as

* Work carried out during the tenure of an ERCIM “Alain Bensoussan” Fellowship Programme. This Programme is supported by the Marie Curie Co-funding of Regional, National and International Programmes (COFUND) of the European Commission.

** Research partially supported by MIUR PRIN 20089M932N project: “Innovative and multi-disciplinary approaches for constraint and preference reasoning”.

---


well as what operations are allowed on given objects. In this paper, when an agent \( A_1 \) adds a piece of information to the store, i.e., a constraint \( c \), it specifies also the confidentiality and integrity rights \([9]\) on that constraint, for each agent \( A_i \) participating to the protected computation. For instance, how much of \( c \) the agent \( A_3 \) may remove from the store (i.e. the retract rights) and how much of \( c \) the agent \( A_2 \) may query with an ask operation (i.e. the ask rights).

We use control mechanisms in order to guarantee some form of security and privacy on the shared store of constraints. However, since we work on the semiring-based formalism \([3]\), our checks are focused on the quantitative, rather than qualitative, point of view, differently from previous works on Linda \([8,5,7]\). In fact, our approach is able to set “how much” of the current store each agent can retract or ask. Therefore, also the rights, together with the information they are applied on (i.e., soft constraints) are soft, in the sense they may concern “part” of the added information. In a crisp vision, if \( c_1 \) is added to the store, it is possible to prevent only the removal of the entire \( c_1 \), but not part of it. When an agent add a constraint to the store by performing a tell action, it also specifies the rights that all the other agents have on that constraint. We define three kinds of rights: the tell rights, stating how much the added constraint can be “worsened” by the other agents, the ask rights, which specify how much of the constraint can be “read” by each agent and the retract rights, describing how much of the added constraint can be removed via a retract action. The tell and retract rights can be classified as integrity rights \([9]\), while the ask rights are classified as confidentiality rights \([9]\). In Def. 1 we define the tell, ask and retract rights.

**Definition 1 (Tell, Ask and Retract Rights).** Let \( n \) be the number of agents participating to the concurrent computation. **Tell rights.** Each constraint \( c_k \) added to the store is associated with a vector \( R_t = (c_{t1}, c_{t2}, \ldots, c_{tn}) \). \( c_{ti} \) represents the tell right imposed on agent \( A_i \). In particular, \( c_{t1} \) represents how much the agent \( A_1 \) can add (with a tell operation) to the constraint \( c_k \), that is how much \( A_1 \) can worsen \( c_k \). **Ask rights.** Each constraint \( c_k \) added to the store is associated with a vector \( R_a = (c_{a1}, c_{a2}, \ldots, c_{an}) \). \( c_{ai} \) represents the ask right imposed on agent \( A_i \). In particular, \( c_{a1} \) represents how much of the added \( c_k \) constraint can be read (with an ask operation in the common store) by agent \( A_1 \). **Retract rights.** Each constraint \( c_k \) added to the store is associated with a vector \( R_r = (c_{r1}, c_{r2}, \ldots, c_{rn}) \). \( c_{ri} \) represents the retract rights imposed on agent \( A_i \). In particular, \( c_{r1} \) represents how much of \( c_k \) can be removed (with a retract operation) by agent \( A_1 \).

We suppose that each agent knows the name (and, consequently, also the number) of the other agents participating to the secure computation on the shared store. This is a general premise for a secure computation, as for example given in Operating Systems Primitives. Moreover, also in the other references in literature an identifier is defined for each entity whose computation is controlled \([5]\). Supposing to know the number of agents at the beginning of the computation is a common practice in many security-related fields, as the execution of multiple threads on the same shared memory. We propose NSCCP as a language to enforce a secure access over general shared resources, checking if
quantitative rights over them are respected, e.g., "Peter may not eat more than 10% of the birthday cake". Moreover, we can suppose that the names of agents are instead names of (security) classes each agent belongs to. The rights of each class are then shared by all the included agents; in this way it is not necessary to set the rights for each single agent, or even to know their number.

With an abuse of notation we define the composition operation of rights as \( \mathcal{W} = \mathcal{R} \otimes \mathcal{R} \), where \( \mathcal{W} \) models the new rights in the computation state after the update, while \( \mathcal{R} \) represents the new rights that modify the state (parameter of the \( \text{tell} \) action in Fig. 1). \( \mathcal{W} = \mathcal{R} \otimes \mathcal{R} \) is implemented with equations (1) \( \forall i. \mathcal{W}_i[i] = \mathcal{R}_i[i] \otimes \mathcal{R}_i[i] \), (2) \( \forall i. \mathcal{W}_a[i] = \mathcal{R}_a[i] \otimes \mathcal{R}_a[i] \) and (3) \( \forall i. \mathcal{W}'_i[i] = \mathcal{R}_i[i] \otimes \mathcal{R}_i[i] \) (i.e. respectively \( \text{tell} \), \( \text{ask} \) and \( \text{retract} \) rights): for example, if we have two agents \( A_1 \) and \( A_2 \), we use the Weighted semiring \( (\aleph^*, \min, +, \infty, 0) \) and \( \mathcal{R}, \mathcal{R} \) are: \( \mathcal{R} = (\mathcal{R}_1 = (x, 5, x + y), \mathcal{R}_a = (y, x, 1), \mathcal{R}_e = (x, z, 2)) \) \( \mathcal{R} = (\mathcal{R}_1 = (y, x, 3), \mathcal{R}_a = (\mathbf{1}, \mathbf{1}, \mathbf{1}), \mathcal{R}_e = (\mathbf{1}, x, 6)) \) then the \( \mathcal{W}' \) composition of rights is given by \( \mathcal{W}' = \mathcal{R} \otimes \mathcal{R} = (\mathcal{W}_r' = (x + y, x + 5, x + y + 3), \mathcal{W}_a' = (y, x, 1), \mathcal{W}_e' = (x, x + z, 8)) \).

The Secure NSCCP Language. Given a soft constraint system \([3]\), in Fig. 1 we present the syntax of the secure NSCCP language \([2]\), which can be used in a secure coordination of agents. 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.

\[
\begin{align*}
P &::= \quad F \cdot A \\
F &::= \quad p(Y) :: A \mid F \cdot F \\
A &::= \quad \text{see} \mid \text{fail} \mid \text{success} \mid \text{tell}(c, \mathcal{R}) \Rightarrow A \mid \text{retract}(c) \Rightarrow A \mid E \mid A \cdot A \mid \exists x. A \mid p(Y) \\
E &::= \quad \text{ask}(c) \Rightarrow A \mid E + E
\end{align*}
\]

Fig. 1: Syntax of the NSCCP language.

The difference w.r.t. \([4]\) is that the \( \text{tell} \) action has a new parameter (in addition to \( c \)), that is the \( \mathcal{R} \) rights. When executing \( \text{tell}(c, \mathcal{R}) \), it is not obviously possible to quantitatively impose more rights on \( c \) than \( c \) itself: therefore, the syntactic conditions on \( \mathcal{R} \) when writing NSCCP programs are that \( \forall i. c + \mathcal{R}_i[i], \ c + \mathcal{R}_a[i], \ c + \mathcal{R}_e[i] \).

To give an operational semantics to our language we describe an appropriate transition system \( (\Gamma, T, \rightarrow) \) where \( \Gamma \) is a set of possible configurations, \( T \subseteq \Gamma \) is the set of \textit{terminal} configurations and \( \rightarrow \subseteq \Gamma \times T \) is a binary relation between configurations. The set of configuration is \( \Gamma = \{ (A, c, \mathcal{R}) \} \) where \( c \in C \) and \( \mathcal{R} \) is the matrix of rights, while the set of terminal configuration is instead \( T = \{ \text{success}, \sigma, \mathcal{R} \} \). To remember also the rights, we need to extend the representation of a computation state in NSCCP in Def. 2.

**Definition 2 (Computation States).** The state of a computation in NSCCP is represented by the triple \( (A, c, \mathcal{R}) \), where \( A \) is the description of the agent still to be executed, \( c \) is the constraint store, and \( \mathcal{R} \) is the set of the rights on the constraints. \( \mathcal{R} \) is initialized as \( \forall i. \mathcal{R}_i[i] = \emptyset, \mathcal{R}_a[i] = \emptyset, \mathcal{R}_e[i] = \emptyset \).
In Fig. 2 we describe the operational semantics of secure NSCCP. A full explanation of the rules is given in [2]. In this paper we add rule R4; with this rule we are able to create a new agent in parallel with the other already being executed. The “body” of the new agent is described by one of the procedures defined in the declaration section, as presented in Fig. 1: in the precondition of the rule, \( p(Y) :: B \in F \). The creating agent can pass to the son a part of his right, and at most all of his rights. These rights are not revoked from the creator.

References