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

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

[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 $\Re_t = \langle c_{t_1}, c_{t_2}, \dots c_{t_n}\rangle$. c_{t_i} represents the* tell right *imposed on agent A_i. In particular, c_{t_i} represents how much the agent A_i can add (with a* tell *operation) to the constraint c_k, that is how much A_i can worsen c_k.* ***Ask rights****. Each constraint c_k added to the store is associated with a vector $\Re_a = \langle c_{a_1}, c_{a_2}, \dots c_{a_n}\rangle$. c_{a_i} represents the* ask right *imposed on agent A_i. In particular, c_{a_i} represents how much of the added c_k constraint can be read (with an* ask *operation in the common store) by agent A_i.* ***Retract rights****. Each constraint c_k added to the store is associated with a vector $\Re_r = \langle c_{r_1}, c_{r_2}, \dots c_{r_n}\rangle$. c_{r_i} represents the* retract rights *imposed on agent A_i. In particular, c_{r_i} represents how much of c_k can be removed (with a* retract *operation) by agent A_i.*

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 $\Re' = \Re \otimes \bar{\Re}$, where $\Re'$ models the new rights in the computation state after the update, while $\bar{\Re}$ represents the new rights that modify the state (parameter of the *tell* action in Fig. 1). $\Re' = \Re \otimes \bar{\Re}$ is implemented with equations $(1)\forall i. \Re'_t[i] = \Re_t[i] \otimes \bar{\Re}_t[i]$, $(2)\forall i. \Re'_a[i] = \Re_a[i] \otimes \bar{\Re}_a[i]$ and $(3)\forall i. \Re'_r[i] = \Re_r[i] \otimes \bar{\Re}_r[i]$ (i.e. respectively *tell, ask* and *retract rights*): for example, if we have two agents A_1 and A_2, we use the *Weighted* semiring $\langle R^+, min, +, \infty, 0\rangle$ and $\Re$, $\bar{\Re}$ are: $\Re = (\Re_t = \langle x, \bar{5}, x + y\rangle, \Re_a = \langle y, x, \bar{\mathbf{1}}\rangle, \Re_r = \langle x, z, \bar{2}\rangle)$ $\bar{\Re} = (\bar{\Re}_t = \langle y, x, \bar{3}\rangle, \bar{\Re}_a = \langle \bar{\mathbf{1}}, \bar{\mathbf{1}}, \bar{\mathbf{1}}\rangle, \bar{\Re}_r = \langle \bar{\mathbf{1}}, x, \bar{6}\rangle)$ then the $\Re'$ composition of rights is given by $\Re' = \Re \otimes \bar{\Re} = (\Re'_t = \langle x + y, x + \bar{5}, x + y + \bar{3}\rangle, \Re'_a = \langle y, x, \bar{\mathbf{1}}\rangle, \Re'_r = \langle x, x + z, \bar{8}\rangle)$.

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.

$$
\begin{array}{rcl}
P & ::= & F.A \\
F & ::= & p(Y) :: A \mid F.F \\
A & ::= & secfail \mid success \mid tell(c, \bar{\Re}) \rightarrowtail A \mid retract(c) \rightarrowtail A \mid E \mid A \| A \mid \exists x.A \mid p(Y) \\
E & ::= & ask(c) \rightarrowtail A \mid E + E
\end{array}
$$

Fig. 1: Syntax of the *NSCCP* language.

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

To give an operational semantics to our language we describe an appropriate transition system $\langle \Gamma, T, \rightarrow\rangle$ where Γ is a set of possible configurations, $T \subseteq \Gamma$ is the set of *terminal* configurations and $\rightarrow \subseteq \Gamma \times T$ is a binary relation between configurations. The set of configuration is $\Gamma = \{\langle A, \sigma, \Re \rangle\}$ where $\sigma \in C$ and $\Re$ is the matrix of rights, while the set of terminal configuration is instead $T = \{\langle success, \sigma, \Re\rangle\}$. 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 $\langle A, \sigma, \Re\rangle$, where A is the description of the agent still to be executed, σ is the constraint store, and $\Re$ is the set of the rights on the constraints. $\Re$ is initialized as $\forall i. \Re_t[i] = \emptyset, \Re_a[i] = \emptyset, \Re_r[i] = \emptyset$.*

$$\mathbf{R1}\ \frac{\sigma \neq \emptyset \quad \mathfrak{R}_t[i] \vdash c \quad \mathfrak{R}_t[i] = \mathfrak{R}_t[i] \odiv c \quad \mathit{check}(\sigma \otimes c)_{\rightarrowtail}}{\langle \mathrm{tell}^i(c, \bar{\mathfrak{R}}) \rightarrowtail A, \sigma, \mathfrak{R} \rangle \longrightarrow \langle A, \sigma \otimes c, \mathfrak{R} \otimes \bar{\mathfrak{R}} \rangle}$$

$$\mathbf{R2}\ \frac{\sigma = \emptyset \quad \mathfrak{R}_t[i] = \mathfrak{R}_t[i] \odiv c \quad \mathit{check}(\sigma \otimes c)_{\rightarrowtail}}{\langle \mathrm{tell}^i(c, \bar{\mathfrak{R}}) \rightarrowtail A, \sigma, \mathfrak{R} \rangle \longrightarrow \langle A, \sigma \otimes c, \mathfrak{R} \otimes \bar{\mathfrak{R}} \rangle}$$

$$\mathbf{R3}\ \frac{\mathfrak{R}_r[i] \vdash c \quad \mathfrak{R}'_r[i] = \mathfrak{R}_r[i] \odiv c \quad \sigma \vdash c \quad \sigma' = \sigma \odiv c \quad \mathit{check}(\sigma \odiv c)_{\rightarrowtail}}{\langle \mathit{retract}^i(c) \rightarrowtail A, \sigma, \mathfrak{R} \rangle \longrightarrow \langle A, \sigma', \mathfrak{R}' \rangle}$$

$$\mathbf{R4}\ \frac{\mathfrak{R}_t[i] \vdash \bar{\mathfrak{R}}_t \quad \mathfrak{R}_a[i] \vdash \bar{\mathfrak{R}}_a \quad \mathfrak{R}_r[i] \vdash \bar{\mathfrak{R}}_r \quad p(Y) :: B \in F \quad \mathit{check}(\sigma)_{\rightarrowtail}}{\langle \mathit{execp}^i(p(Y), \bar{\mathfrak{R}})) \rightarrowtail A, \sigma, \mathfrak{R} \rangle \longrightarrow \langle A \parallel B, \sigma, \mathfrak{R} \cup \bar{\mathfrak{R}} \rangle}$$

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

$$\mathbf{R6}\ \frac{\mathfrak{R}_a[i] \vdash c \quad \sigma \vdash c \quad \mathit{check}(\sigma)_{\rightarrowtail}}{\langle \mathrm{ask}^i(c) \rightarrowtail A, \sigma, \mathfrak{R} \rangle \longrightarrow \langle A, \sigma, \mathfrak{R} \rangle}$$

$$\mathbf{R7}\ \frac{\mathfrak{R}_a[i] \vdash c \quad \sigma \nvdash c \quad \mathit{check}(\sigma)_{\rightarrowtail}}{\langle \mathit{nask}(c) \rightarrowtail A, \sigma \rangle \longrightarrow \langle A, \sigma \rangle}$$

$$\mathbf{R8}\ \frac{\langle A, \sigma, \mathfrak{R} \rangle \longrightarrow \langle A', \sigma', \mathfrak{R}' \rangle}{\begin{array}{c}\langle A \parallel B, \sigma, \mathfrak{R} \rangle \longrightarrow \langle A' \parallel B, \sigma', \mathfrak{R}' \rangle \\ \langle B \parallel A, \sigma, \mathfrak{R} \rangle \longrightarrow \langle B \parallel A', \sigma', \mathfrak{R}' \rangle\end{array}}$$

$$\mathbf{R9}\ \frac{\langle A, \sigma, \mathfrak{R} \rangle \longrightarrow \langle \mathit{success}, \sigma', \mathfrak{R}' \rangle}{\begin{array}{c}\langle A \parallel B, \sigma, \mathfrak{R} \rangle \longrightarrow \langle B, \sigma', \mathfrak{R}' \rangle \\ \langle B \parallel A, \sigma, \mathfrak{R} \rangle \longrightarrow \langle B, \sigma', \mathfrak{R}' \rangle\end{array}}$$

$$\mathbf{R10}\ \frac{\langle A[x/y], \sigma, \mathfrak{R} \rangle \longrightarrow \langle B, \sigma', \mathfrak{R}' \rangle}{\langle \exists x.A, \sigma, \mathfrak{R} \rangle \longrightarrow \langle B, \sigma', \mathfrak{R}' \rangle} \quad y\ \mathit{fresh}$$

$$\mathbf{R11}\ \frac{\langle A, \sigma, \mathfrak{R} \rangle \longrightarrow \langle B, \sigma', \mathfrak{R}' \rangle}{\langle p(Y), \sigma, \mathfrak{R} \rangle \longrightarrow \langle B, \sigma', \mathfrak{R}' \rangle} \quad p(Y) :: A \in F$$

Fig. 2: The transition system for *NSCCP*.

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 F, 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

1. Bella, G., Bistarelli, S.: Soft constraint programming to analysing security protocols. TPLP 4(5-6), 545–572 (2004)
2. Bistarelli, S., Campli, P., Santini, F.: A secure coordination of agents with nonmonotonic Soft Concurrent Constraint Programming. In: Proceedings of the ACM Symposium on Applied Computing, SAC 2012. pp. 1551–1553. ACM (2012)
3. Bistarelli, S., Montanari, U., Rossi, F.: Soft concurrent constraint programming. ACM Trans. Comput. Logic 7(3), 563–589 (2006)
4. Bistarelli, S., Santini, F.: A nonmonotonic soft concurrent constraint language to model the negotiation process. to appear in Fundamenta Informaticae (2011)
5. Gorrieri, R., Lucchi, R., Zavattaro, G.: Supporting secure coordination in SecSpaces. Fundam. Inform. 73(4), 479–506 (2006)
6. Sandhu, R., Samarati, P.: Access control: Principles and practice. IEEE Communications 32(9), 40–48 (1994)
7. Udzir, N.I., Wood, A.M., Jacob, J.L.: Coordination with multicapabilities. Sci. Comput. Program. 64(2), 205–222 (2007)
8. Vitek, J., Bryce, C., Oriol, M.: Coordinating processes with secure spaces. Sci. Comput. Program. 46(1-2), 163–193 (2003)
9. Whitman, M.E., Mattord, H.J.: Principles of Information Security. Course Technology Press, Boston, MA, USA, 3rd edn. (2007)