Securely Accessing Shared Resources with Concurrent Constraint Programming

S. Bistarelli [1,2], F. Santini [3,1]

[1]Dipartimento di Matematica e Informatica, Perugia

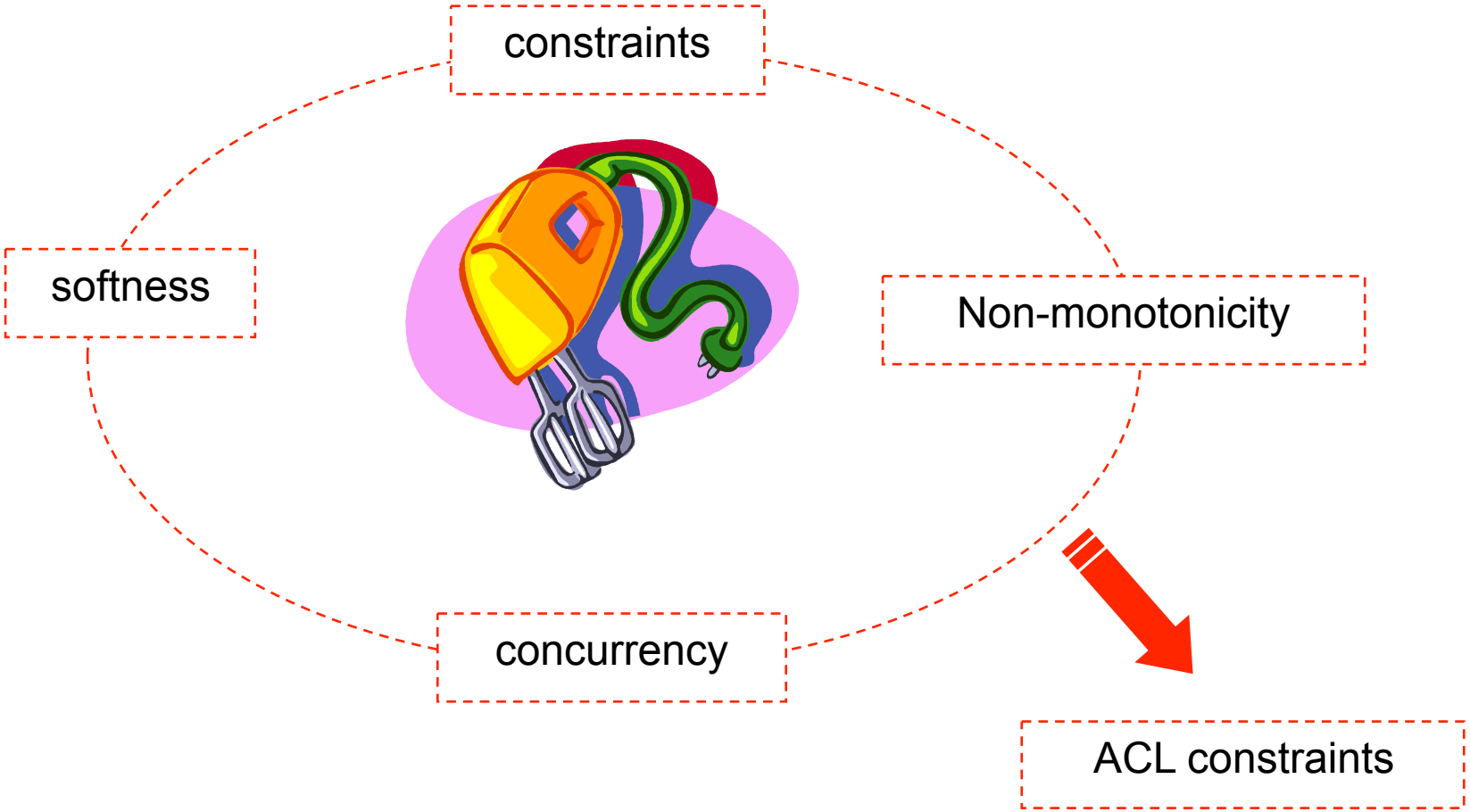
[2]Istituto di Informatica e Telematica CNR, Pisa

[3]INRIA-Rocquencourt, France

Motivations

- A language for agent coordination
  - With control on the actions they perform
  - guaranteeing security properties (integrity and confidentiality)
- Based on (soft) constraints
  - Each resource is a constraint with a cost/preference **quantitative** label
  - Fine grained level of activity (not only a complete token of information/resource but also a **partial** modification/check of the store)
- Extending (nonmonotone) cc
  - Possibility for agents to interact adding and checking resource status, but also **removing** and **modifiying** resources

The basic idea



ACL constraints

- An ACL is a list of permissions attached to an object
  - it specifies users/system processes, objects, and the allowed operations.
- Our ACL constraints
  - when an agent A1 adds a piece of information to the store (tell c), it specifies also the confidentiality and integrity rights on that constraint, for each agent Ai participating to the protected computation.
    - how much of c the agent A3 can remove from the store (retract) and
    - how much of c the agent A2 can query with an ask operation (ask).
  - e.g. "**Peter may not eat more than 10% of the birthday cake**".

Summary

- **The starting frameworks**
  - **Semirings and soft constraints [BMR JACM97]**
  - **Concurrent constraints [Saraswat POPL87]**
  - **Soft concurrent constraints [BMR TOCL2006]**
  - **Nonmonotonic soft cc [BS FundInf2011]**
- ACL constraints and security primitives (execp)
- An example
- Conclusions and related works

C-semirings

$$\langle A, +, \times, 0, 1 \rangle$$

- A semiring is a tuple
  - The + is commutative, associative and 0 is its unit element
  - The $\times$ is associative, distributes over +, 1 is its unit element and 0 is its absorbing element
- A c-semiring is a semiring such that
  - + is idempotent, 1 is its absorbing element and $\times$ is commutative
- Consider the relation $\leq_S$ over *A* such that $a \leq_S b$ iif $a + b = b$
  - $\leq_S$ is a partial order
  - + and $\times$ are monotone
  - 0 is its minimum and 1 its maximum
  - $\langle A, \leq_S \rangle$ is a complete lattice and, for all $a, b \in A, a + b = lub(a, b)$

+ to find the best cost and $\times$ to compose the costs

C-semirings examples

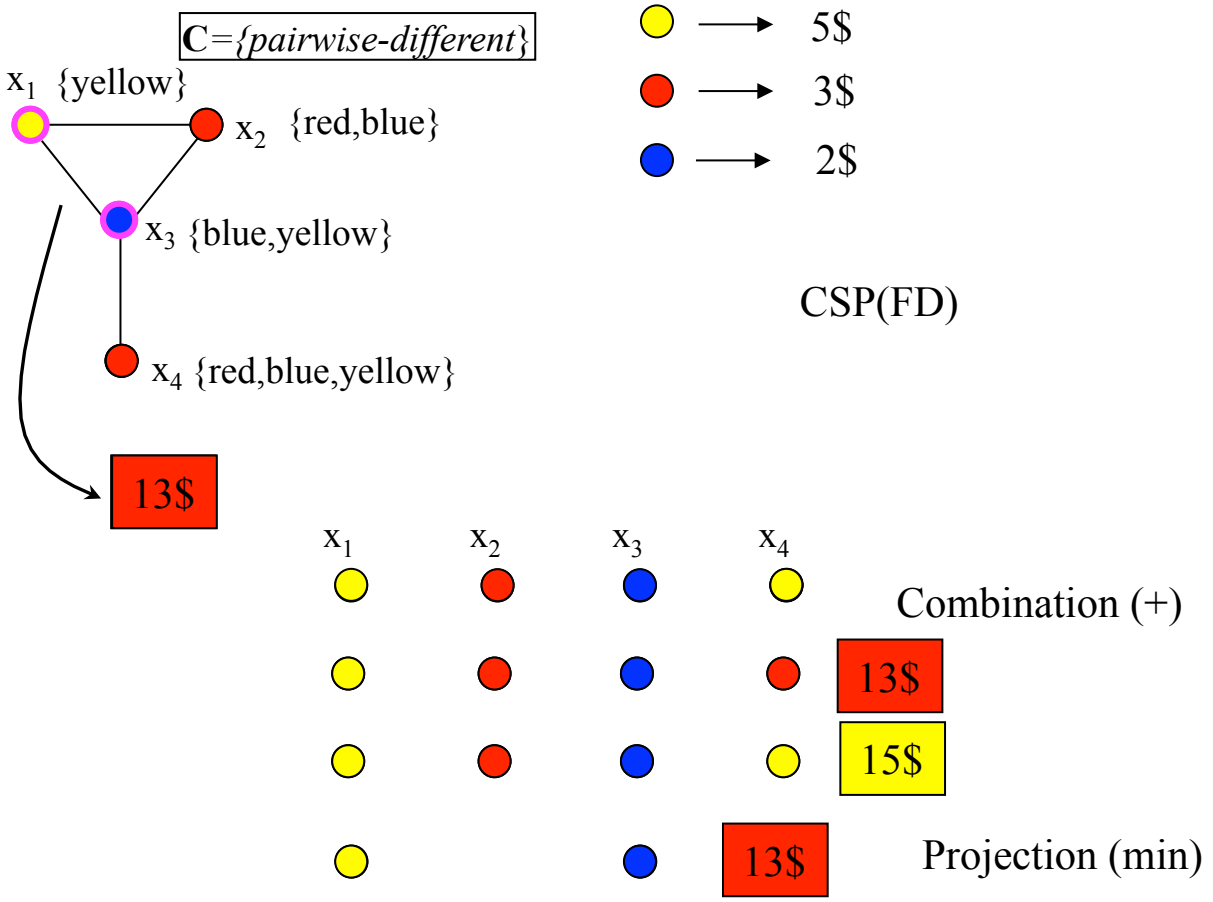
- Weighted $\langle R^{+}, min, +, \infty, 0\rangle$
  - Money cost: link maintenance or billing criteria
- Fuzzy $\langle [0,1], max, min, 0, 1\rangle$
  - It can be used to maximize bandwidth
- Probabilistic $\langle [0,1], max, \times, 0, 1\rangle$
  - Successful delivery of the packet
- Attributes/Rights $\langle \mathcal{P}(A), \cup, \cap, \emptyset, A\rangle$
  - Security, time slots, etc

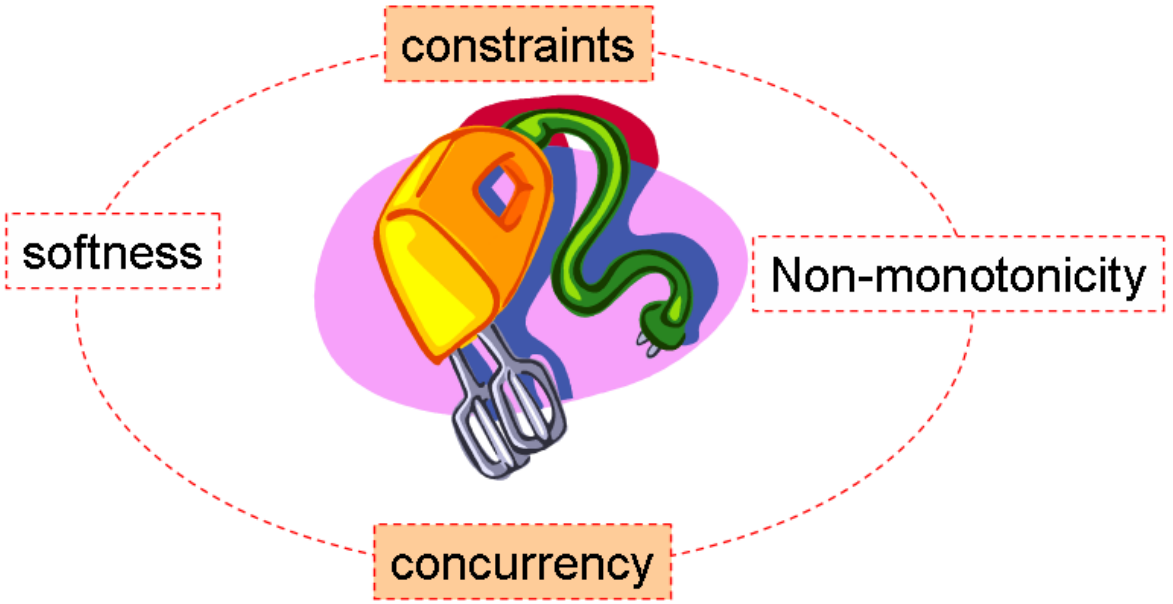
The Cartesian product of two semirings is still a semiring: multicriteria optimization

All that can be represented with a semiring, can be cast in this framework as a QoS parameter

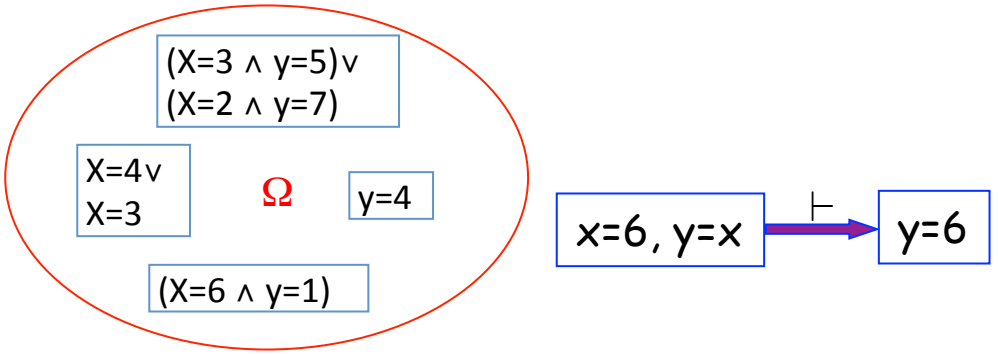
$$S_{Network} = \langle\langle [0,1], \mathcal{R}^{+}\rangle, \langle max, min\rangle, \langle min, +\rangle, \langle 0, +\infty\rangle, \langle +\infty, 0\rangle\rangle$$

The semiring based constraint solving framework



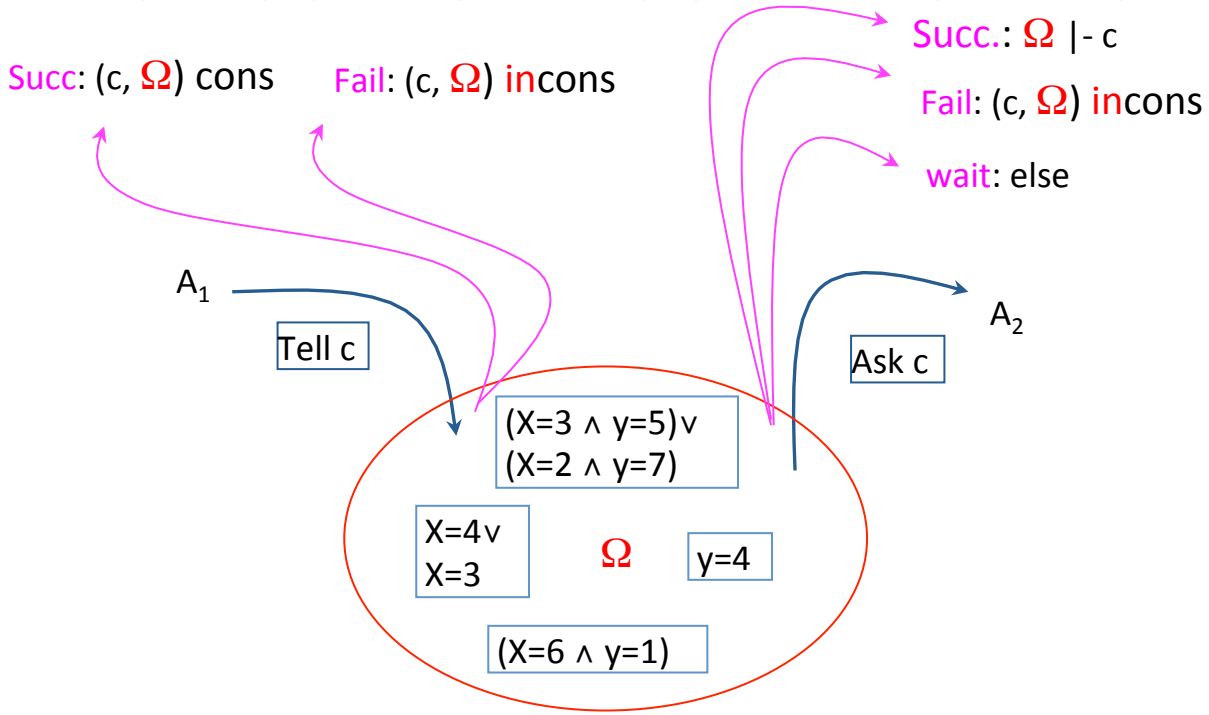

constraints
softness
Non-monotonicity
concurrency

Concurrent constraints



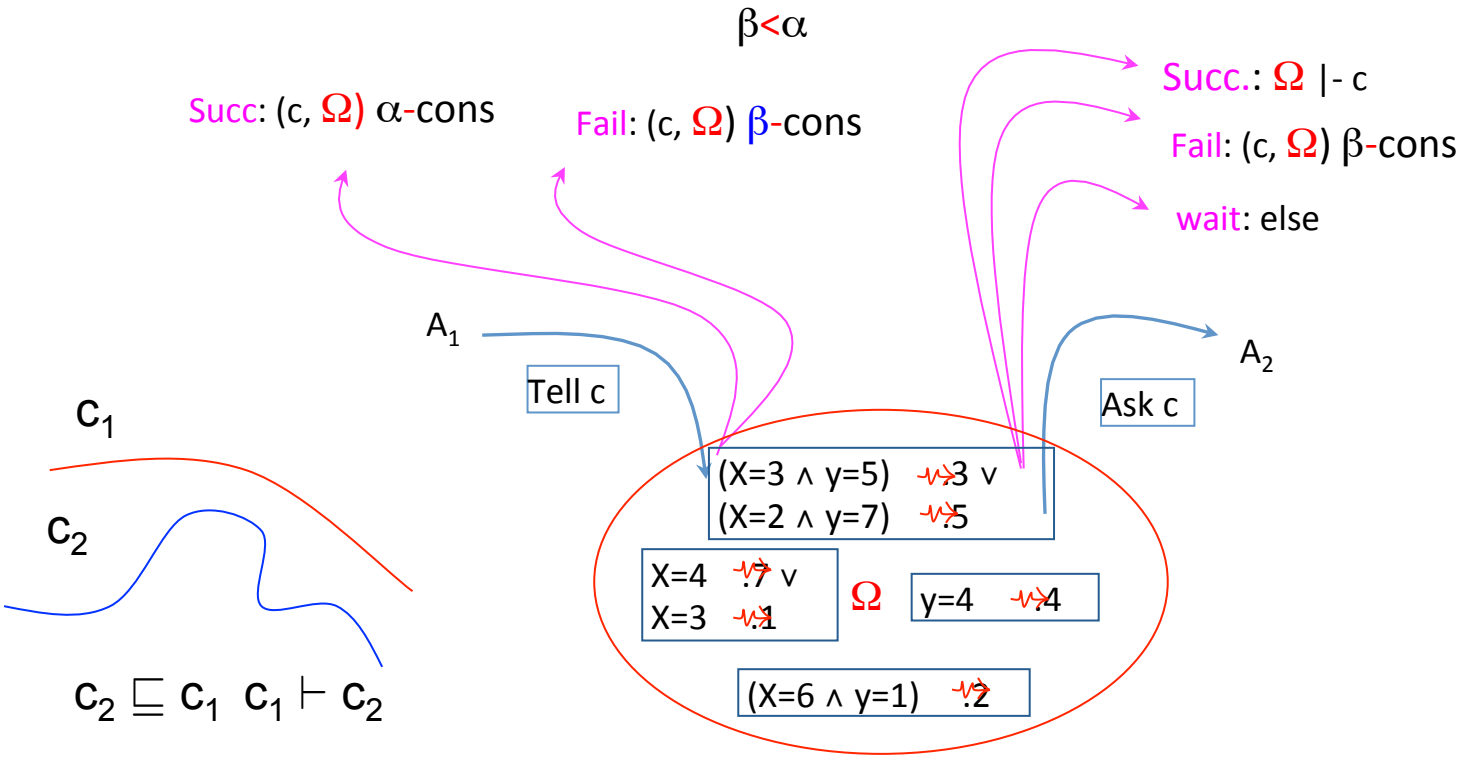
- Constraint system as information systems
  - A set D of primitive constraints (or tokens)
    - u ⊢ P for all P in u (reflexivity)
    - u ⊢ v, v ⊢ z, u ⊢ z (transitivity)

Concurrent constraints



- Constraint system as information systems
  - A set D of primitive constraints (or tokens)
    - $u \vdash P$ for all P in u (reflexivity)
    - $u \vdash v, v \vdash z, u \vdash z$ (transitivity)

Soft cc: the idea



- Changing the cut level we obtain different results

Soft Constraints Systems

- A constraint with an associated preference
  - V= variables, D = domains and
  - where *A* is the semiring set
  - Where is an assignment
- is defined as
- is defined as
- is defined as

$S = \langle A, +, \times, 0, 1 \rangle$

$C : (V \to D) \to A$

$\eta : V \to D$

$\otimes : C \times C \to C$

$\oplus : C \times C :\to C$

$\oplus\!\!\!\!- : C \times C \to C$

$(c_1 \otimes c_2)\eta = c_1\eta \times c_2\eta$

$(c_1 \oplus c_2)\eta = c_1\eta + c_2\eta$

$(c_1 \oplus\!\!\!\!- c_2)\eta = c_1\eta \div c_2\eta$

BG@ECAI2006

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

$c_2 : \{x\} \to \mathbb{N} \to \mathbb{R}^+ \quad \text{s.t. } c_4(x) = x + 5$

$c_3 : \{x\} \to \mathbb{N} \to \mathbb{R}^+ \quad \text{s.t. } c_2(x) = 2x + 8$

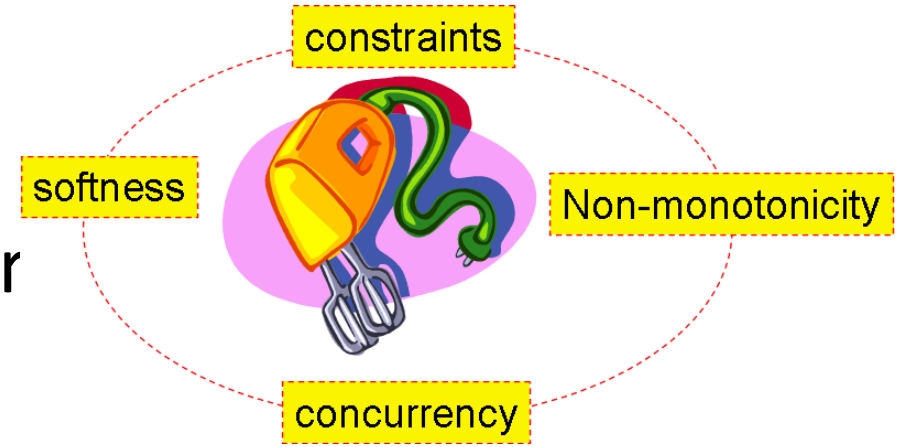
$c_3 = c_1 \otimes c_2 \qquad c_1 = c_3 \oplus\!\!\!\!- c_2$

weighted

$\langle R^+, min, +, \infty, 0 \rangle$

$c_2 \vdash c_1$

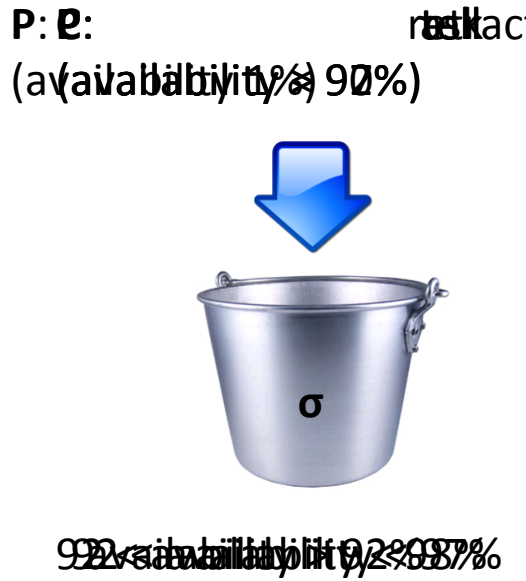
Nonmonotonic soft cc



- Soft Constraints + concur
  - +
- Non-monotonicity
  - To revise opinions (update)
  - To relax requirements (retract)
  - To check entailment but also non-entailment (nask)

nonmonotonic SCC

- A centralized store where to compose the requirements of the entities
- A formal language:
  - *tell* and *ask*
  - *retract* and *nask*



Summary

- The language
  - Semirings and soft constraints
  - Concurrent constraints
  - Soft concurrent constraints
  - Nonmonotonic soft cc
- **ACL constraints and security primitives (execp)**
- An example
- Conclusions and related works

the Tell/Ask/Retract Rights

- When an agent tell a constraint, it also specifies 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.

➘Able to set "how much" of the current store each agent can retract or ask.

➘In a crisp vision of access control, if c1 is added to the store, it would be possible to prevent only the removal of the entire c1

ACL soft constraints

- *Both* resources and rights are seen as soft constraints
- Each entry in a typical ACL specifies a subject and an operation and follows the structure:

  (object identity, user identity) -> permitted operations.
- The matrix stores the ACLs at each step of the computation (becomes part of the state)
- The matrix represent what each Agent can do on the store

	$\mathfrak{R}_t$ (Tell)	$\mathfrak{R}_a$ (Ask)	$\mathfrak{R}_r$ (Retract)
$Agent_1$	$\mathfrak{R}_t[1]$	$\mathfrak{R}_a[1]$	$\mathfrak{R}_r[1]$
$Agent_2$	$\mathfrak{R}_t[2]$	$\mathfrak{R}_a[2]$	$\mathfrak{R}_r[2]$
...	...	...	...
$Agent_n$	$\mathfrak{R}_t[n]$	$\mathfrak{R}_a[n]$	$\mathfrak{R}_r[n]$

ACL soft constraints

- The rights are specified when we add information to the store, with the tell

$$tell(c, \bar{\mathfrak{R}}) \qquad \forall i.\, c \vdash \bar{\mathfrak{R}}_t[i], c \vdash \bar{\mathfrak{R}}_a[i], c \vdash \bar{\mathfrak{R}}_r[i]$$

- New rights are composed with the actual matrix of rights

$$\forall i.\, \mathfrak{R}'_t[i] = \mathfrak{R}_t[i] \otimes \bar{\mathfrak{R}}_t[i],\ \mathfrak{R}'_a[i] = \mathfrak{R}_a[i] \otimes \bar{\mathfrak{R}}_a[i],\ \mathfrak{R}'_r[i] = \mathfrak{R}_r[i] \otimes \bar{\mathfrak{R}}_r[i]$$

The Language

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

Operational Semantics

R1 $$\frac{\sigma \neq \emptyset \qquad \Re_t[i] \vdash c \qquad \Re_t[i] = \Re_t[i] \oplus c \qquad check(\sigma \otimes c)_{\rightarrowtail}}{\langle \text{tell}^i(c, \bar{\Re}) \rightarrowtail A, \sigma, \Re \rangle \rightarrow \langle A, \sigma \otimes c, \Re \otimes \bar{\Re} \rangle}$$ **Tell 1**

R2 $$\frac{\sigma = \emptyset \qquad \Re_t[i] = \Re_t[i] \oplus c \qquad check(\sigma \otimes c)_{\rightarrowtail}}{\langle \text{tell}^i(c, \bar{\Re}) \rightarrowtail A, \sigma, \Re \rangle \rightarrow \langle A, \sigma \otimes c, \Re \otimes \bar{\Re} \rangle}$$ **Tell 2**

R3 $$\frac{\Re_r[i] \vdash c \quad \Re'_r[i] = \Re_r[i] \ominus c \quad \sigma \vdash c \quad \sigma' = \sigma \ominus c \quad check(\sigma \ominus c)_{\rightarrowtail}}{\langle retract^i(c) \rightarrowtail A, \sigma, \Re \rangle \rightarrow \langle A, \sigma', \Re' \rangle}$$ **Retract**

R4 $$\frac{\Re_t[i] \vdash \bar{\Re}_t \quad \Re_a[i] \vdash \bar{\Re}_a \quad \Re_r[i] \vdash \bar{\Re}_r \quad p(Y) :: B \in F \qquad check(\sigma)_{\rightarrowtail}}{\langle execp^i(p(Y), \bar{\Re})) \rightarrowtail A, \sigma, \Re \rangle \rightarrow \langle A \parallel B, \sigma, \Re \cup \bar{\Re} \rangle}$$ **ExecP**

R5 $$\frac{\langle E_j, \sigma, \Re \rangle \rightarrow \langle A_j, \sigma', \Re' \rangle \qquad j \in [1, n]}{\langle \Sigma_{i=1}^{n} E_i, \sigma, \Re \rangle \rightarrow \langle A_j, \sigma', \Re' \rangle}$$ **Nondet**

R6 $\dfrac{\mathfrak{R}_a[i] \vdash c \qquad \sigma \vdash c \qquad check(\sigma)_{\rightarrowtail}}{\langle \mathrm{ask}^i(c) \rightarrowtail A, \sigma, \mathfrak{R} \rangle \rightarrow \langle A, \sigma, \mathfrak{R} \rangle}$ **Ask**

R7 $\dfrac{\mathfrak{R}_a[i] \vdash c \qquad \sigma \nvdash c \qquad check(\sigma)_{\rightarrowtail}}{\langle nask(c) \rightarrowtail A, \sigma \rangle \rightarrow \langle A, \sigma \rangle}$ **Nask**

R8 $\dfrac{\langle A, \sigma, \mathfrak{R} \rangle \rightarrow \langle A', \sigma', \mathfrak{R}' \rangle}{\begin{array}{c}\langle A \parallel B, \sigma, \mathfrak{R} \rangle \rightarrow \langle A' \parallel B, \sigma', \mathfrak{R}' \rangle \\ \langle B \parallel A, \sigma, \mathfrak{R} \rangle \rightarrow \langle B \parallel A', \sigma', \mathfrak{R}' \rangle\end{array}}$ **Parallel 1**

R9 $\dfrac{\langle A, \sigma, \mathfrak{R} \rangle \rightarrow \langle success, \sigma', \mathfrak{R}' \rangle}{\begin{array}{c}\langle A \parallel B, \sigma, \mathfrak{R} \rangle \rightarrow \langle B, \sigma', \mathfrak{R}' \rangle \\ \langle B \parallel A, \sigma, \mathfrak{R} \rangle \rightarrow \langle B, \sigma', \mathfrak{R}' \rangle\end{array}}$ **Parallel 2**

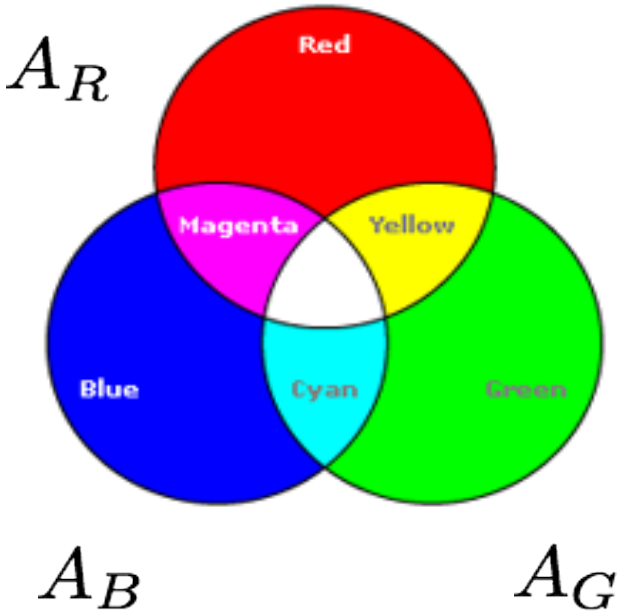
R10 $\dfrac{\langle A[x/y], \sigma, \mathfrak{R} \rangle \rightarrow \langle B, \sigma', \mathfrak{R}' \rangle}{\langle \exists x.A, \sigma, \mathfrak{R} \rangle \rightarrow \langle B, \sigma', \mathfrak{R}' \rangle}$ y *fresh* **Hiding**

R11 $\dfrac{\langle A, \sigma, \mathfrak{R} \rangle \rightarrow \langle B, \sigma', \mathfrak{R}' \rangle}{\langle p(Y), \sigma, \mathfrak{R} \rangle \rightarrow \langle B, \sigma', \mathfrak{R}' \rangle}$ $p(Y) :: A \in F$ **Proc-call**

Summary

- The language
  - Semirings and soft constraints
  - Concurrent constraints
  - Soft concurrent constraints
  - Nonmonotonic soft cc
- ACL constraints and security primitives (execp)
- **An example**
- Conclusions and related works

RGB Monitor example



One common application of the RGB color model is the display of colors on a CRT, LCD or LED displays such as a television, a computer's monitor, or a large scale screen. Each pixel on the screen is built by driving three small and very close but still separated RGB light sources. During digital image processing each pixel can be represented in the computer memory or interface hardware as binary values for the red, green, and blue color components.

•The principal manager (AC) adds color white (i.e., all the possible information) to the store and then creates three separate sub-managers of, respectively, red (AR), green (AG) and blue color (AB). Each of these managers is in charge to modify its color by augmenting or decreasing its intensity, query the intensity of the other colors if they want, but not to change the other two colors.

A_C is the initial color manager

$$execp(P_{Blue}, \Re_{A_B}) \rightarrow^{brown}_{grey} nask(white) \rightarrow^{brown}_{grey} ask(yellow) \rightarrow^{brown}_{yellow} success, black\rangle$$

The lattice of colors (and rights)

The lattice of colors (and rights)

- AC checks the end of the color sequence, that is if yellow is in the store. Moreover, AC checks if the intensity of the brightest and darkest color obtained in the store is above grey and below brown in order to limit the flickering of the screen.

$$A_C :: \langle tell(white, \Re_{A_C}) \xrightarrow{black} execp(P_{Red}, \Re_{A_R}) \xrightarrow{black} execp(P_{Green}, \Re_{A_G}) \xrightarrow{black} execp(P_{Blue}, \Re_{A_B}) \xrightarrow{brown} nask(white) \xrightarrow{brown} ask(yellow) \xrightarrow{brown} success, black\rangle$$

•The matrix of the rights after the creation of AR, AG, AB

	$\Re_t$ (Tell)	$\Re_a$ (Ask)	$\Re_r$ (Retract)
A_C	*black*	*white*	*black*
A_R	*black*	*red*	*black*
A_G	*green*	*magenta*	*green*
A_B	*black*	*white*	*blue*

The description of the three color sub-managers A_R A_G A_B

$$P_{Red} :: ask(red) \rightarrow^{black}_{white} success$$

$$P_{Green} :: retract(green) \rightarrow^{black}_{white} nask(magenta) \rightarrow^{black}_{white} tell(green, \Re_G) \rightarrow^{black}_{white} success$$

$$P_{Blue} :: nask(white) \rightarrow^{black}_{white} retract(blue) \rightarrow^{black}_{white} success$$

•Thematrix of the rights at the end of the parallel computation.

	$\Re_t$ (Tell)	$\Re_a$ (Ask)	$\Re_r$ (Retract)
A_C	*black*	*white*	*black*
A_R	*black*	*red*	*black*
A_G	*black*	*magenta*	*black*
A_B	*black*	*white*	*black*

Summary

- The language
  - Semirings and soft constraints
  - Concurrent constraints
  - Soft concurrent constraints
  - Nonmonotonic soft cc
- ACL constraints and security primitives (execp)
- An example
- **Conclusions and related works**

Related work: Linda

- **Linda** is implemented as a coordination language in which the primitives operate on an ordered sequence of typed data objects called "tuples"; these primitives are added to a sequential language, and interact with a logically global associative memory, called the tuplespace, in which processes store and retrieve tuples. The basic primitives are:
  - **in**, which atomically reads and removes (i.e., consumes) a tuple from tuplespace,
  - **rd** non-destructively reads a tuplespace

Related work: Linda ext.

- **SecSpaces** [Vitex et al.] extends Linda with fine-grained access control using a capability-based system.
- three mechanisms to manage tuple access control [Gorrieri et al.].
  - logical partitions of the shared repository: in this way we can restrict the access to tuples inside a partition, simply by limiting the access to the partition itself.
  - extra information in the tuple which exploit asymmetric cryptography in order, for instance, to

Related work: Linda ext.

- **Klaim** [DeNicola et al.] exploits a standard access control mechanism where permissions describe which operations the agents can perform on the available spaces.
- **MuKlaim** [Pugliese et al.] allows dynamic permission acquisition.
- **Klava** [DeNicola et al.] introduces encrypted messages into the fields of the tuples and the matching rule allows the evaluation of messages encrypted into fields; the encryption of

Conclusions and Future work

- A fine-grained security model to enforce the access control, inspired by Linda
- Information, unlike Linda, **is not crisp (soft constraints)**
- Both information and rights are represented with the same model (soft constraints), again unlike Linda
- Based on the ACL security model, it can enforce Discretionary and Mandatory Access Control principles (MAC and DAC)