Programming Abstractions for Autonomic Computing

Rocco De Nicola
IMT - Institute for Advanced Studies Lucca
Joint work with G. Ferrari, M. Loreti, R. Pugliese, M. Loreti

CINA Kick off Meeting
Pisa, February 4-6, 2013

Outline

Challenges of Ensembles Programming

ascens

Ensembles are software-intensive systems featuring

- massive numbers of components
- complex interactions among components, and with other systems
- operating in open and non-deterministic environments
- dynamically adapting to new requirements, technologies and environmental conditions

From the final report of: IST Coordinated Action InterLink [2007].

Challenges for software development for ensembles

- the dimension of the systems
- the need to adapt to changing environments and requirements
- the emergent behaviour resulting from complex interactions
- the uncertainty during design-time and run-time

ascens

Components and Ensembles

Service components (SCs) and service-component ensembles (SCEs) permit to dynamically structure independent, distributed entities that can cooperate, with different roles, in open and non-deterministic environments

Awareness

Awareness of Service Components is achieved by

- equipping SCs with information about their own state
- enabling SCs to get information on their working environment
- allowing SCs to use this information for restructuring and adapting

Awareness makes SCs adaptable, connectable and composable.

ascens

We aim at at developing linguistic supports for modelling (and programming) the behavior of service components and their ensembles, their interactions, their sensitivity and adaptivity to the environment

SCEL

We aim at designing a specific language with

- programming abstractions necessary for
  - directly representing Knowledge, Behaviors and Aggregations according to specific Policies
  - naturally programming interaction, adaptation and self- and context-awareness
- linguistic primitives with solid semantic grounds
  - To develop logics, tools and methodologies for formal reasoning on systems behavior
  - to establish qualitative and quantitative properties of both the individual components and the ensembles

Key Notions

ascens

We need to enable programmers to model and describe the behavior of service components and their ensembles, their interactions, and their sensitivity and adaptivity to the environment they are working in.

Notions to model

1. The behaviors of components and their interactions
2. The topology of the network needed for interaction, taking into account resources, locations and visibility/reachability issues
3. The environment where components operate and resource-negotiation takes place, taking into account open ended-ness and adaptation
4. The global knowledge of the systems and that of its components
5. The tasks to be accomplished, the properties to guarantee and the constraints to respect.

SCEL: programming abstractions

ascens

The Service-Component Ensemble Language (SCEL) currently provides primitives and constructs for dealing with 4 programming abstractions.

1. Knowledge: to describe how data, information and (local and global) knowledge is managed
2. Behaviours: to describe how systems of components progress
3. Aggregations: to describe how different entities are brought together to form *components*, *systems* and *ensembles*
4. Policies: to model and enforce the wanted evolutions of computations.

1. Knowledge

SCEL is *parametric* wrt the means of managing knowledge that would depend on the specific class of application domains.

Knowledge representation

- Tuples, Records
- Horn Clause Clauses,
- Concurrent Constraints,
- ...

Knowledge handling mechanisms

- Pattern-matching, Reactive Tuple Spaces
- Data Bases Querying
- Resolution
- Constraint Solving
- ...

1. Knowledge (and Adaptation)

ascens

No definite stand is taken about the kind of knowledge that might depend on the application domain. To guarantee adaptivity, we, however, require there be some specific components.

Application data

Used for the progress of the computation.

Control data

Providing information about the environment (e.g. data from sensors) and about the current status (e.g. its position or its battery level).

Knowledge handling mechanisms

- Add information to a knowledge repository
- Retrieve information from a knowledge repository
- Withdraw information from a knowledge repository

ascens

Components behaviors are modeled as a process in the style of process calculi

- Interaction is obtained by allowing processes to access knowledge repositories, possibly of other components
- Adaptation is modeled by retrieving from the knowledge repositories
  - information about the changing environment and the component status
  - the code to execute for reacting to these changes.

Processes

$$P \quad ::= \quad \mathbf{nil} \;\mid\; a.P \;\mid\; P_1 + P_2 \;\mid\; P_1[\,P_2\,] \;\mid\; X \;\mid\; A(\bar{p}) \quad (A(\bar{f}) \triangleq P)$$

The operators have the expected semantics. $P_1[\,P_2\,]$ (Controlled Composition) can be seen as a generalization of the many "parallel compositions" of process calculi. For the meaning of $a.-$, see next.

ascens

Actions

$$a ::= \mathbf{get}(T)@c \mid \mathbf{qry}(T)@c \mid \mathbf{put}(t)@c \mid \mathbf{fresh}(n) \mid \mathbf{new}(\mathcal{I}, \mathcal{K}, \Pi, P)$$

Action Targets

$$c ::= n \mid x \mid \text{self} \mid \textit{ensemble}(?)$$

Rôle of Actions

- manage knowledge repositories by
  - withdrawing information - $\mathbf{get}(T)@c$,
  - retrieving information - $\mathbf{qry}(T)@c$
  - adding information - $\mathbf{put}(t)@c$

  Actions operate on knowledge repository c and use T as a pattern to select knowledge items.
- create new names or new components $\mathcal{I}[\mathcal{K}, \Pi, P]$ - $\mathbf{new}(\mathcal{I}, \mathcal{K}, \Pi, P)$

ascens

Aggregations

describe how different entities are brought together

- Model resource *allocation* and *distribution*
- Reflect the idea of *administrative domains*, i.e. the authority controlling a given set of resources and computing agents.
- are modelled by resorting to the notions of system, component and ensemble.

Systems

$$S \quad ::= \quad C \quad | \quad S_1 \parallel S_2 \quad | \quad (\nu n)S$$

- Single component C (see next slide)
- Parallel composition $\_ \parallel \_$
- Name restriction $\nu n\_$ (to delimit the scope of name n), thus in $S_1 \parallel (\nu n)S_2$, name n is invisible from within S_1

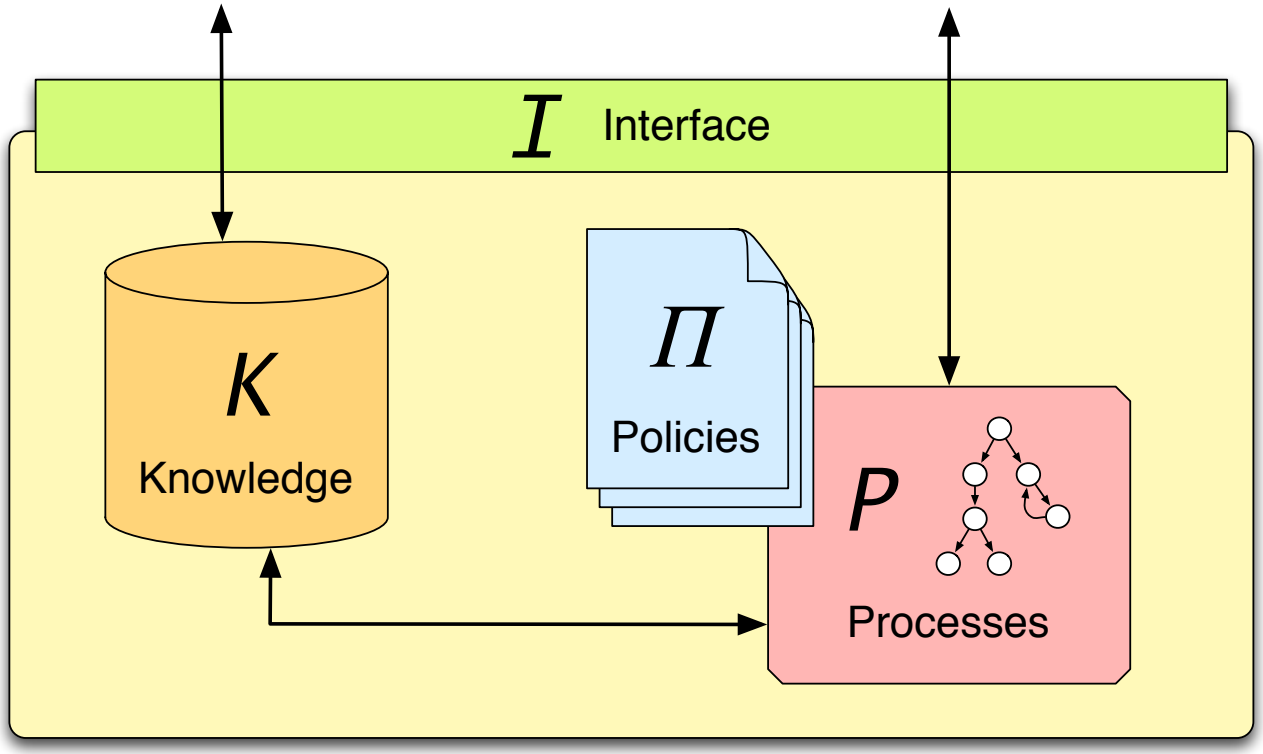
ascens

Components

$$C \quad ::= \quad \mathcal{I}[\mathcal{K}, \Pi, P]$$

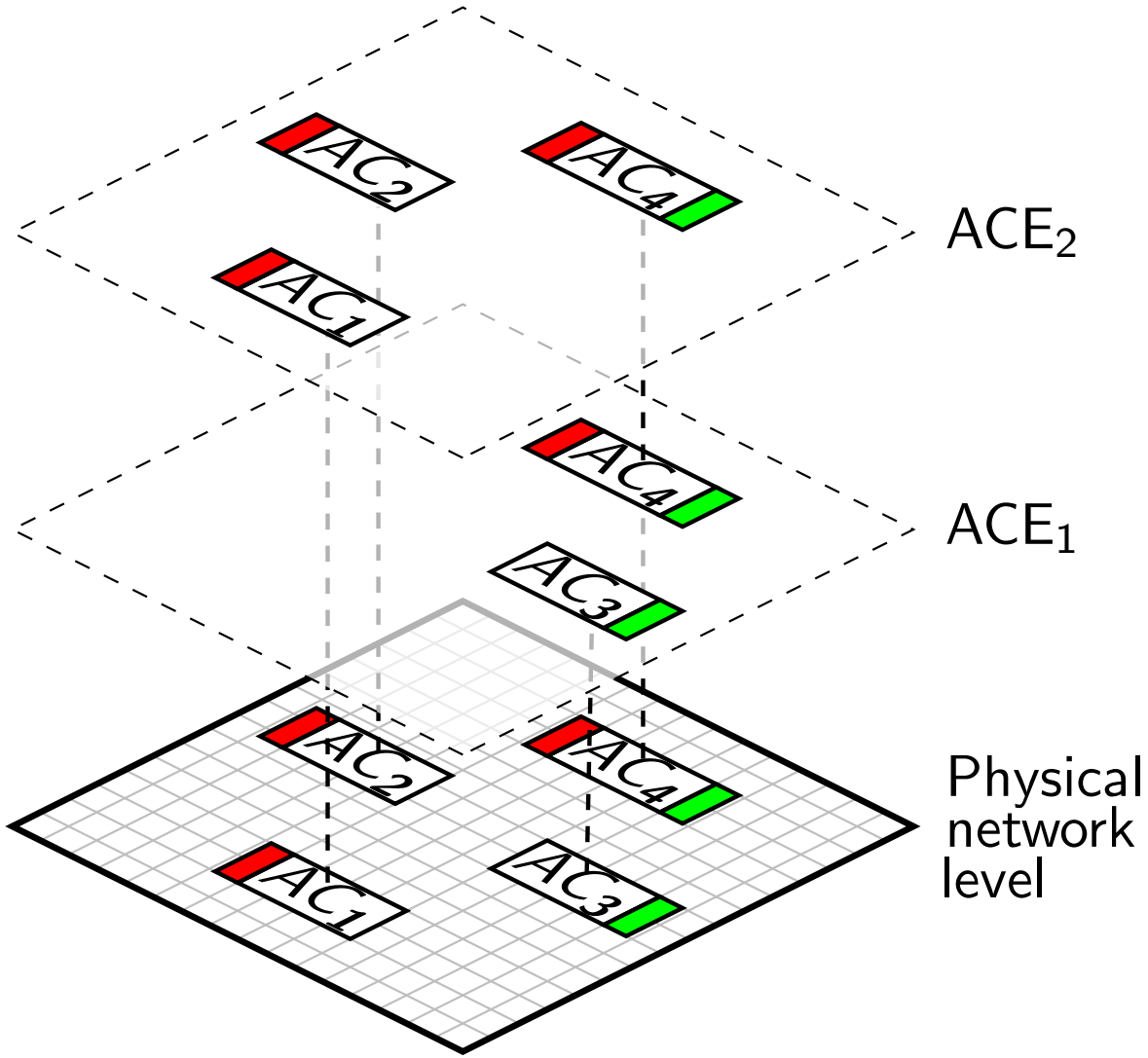
- An interface $\mathcal{I}$ containing information about the component itself. In particular, each component C has attributes:
  - *id*: the name of the component C
- A knowledge manager $\mathcal{K}$ providing control data (i.e. the local and (part of the) global knowledge) and application data; together with a specific knowledge handling mechanism
- A set of policies Π regulating inter-component and intra-component interactions
- A process term P that performs the local computation, coordinates their interaction with the knowledge repository and deals with adaptation and reconfiguration

A SCEL component



Programming Abstractions

Important for improving code productivity


AC2
AC4
AC1
ACE2
AC4
AC3
ACE1
AC2
AC4
AC1
AC3
Physical
network
level

4. Policies

ascens

Policies deal with the way properties of computations are represented and enforced

- Interaction: interaction predicates, ...
- Resource usage: accounting, leasing, ...
- Security: access control, trust, reputation, ...

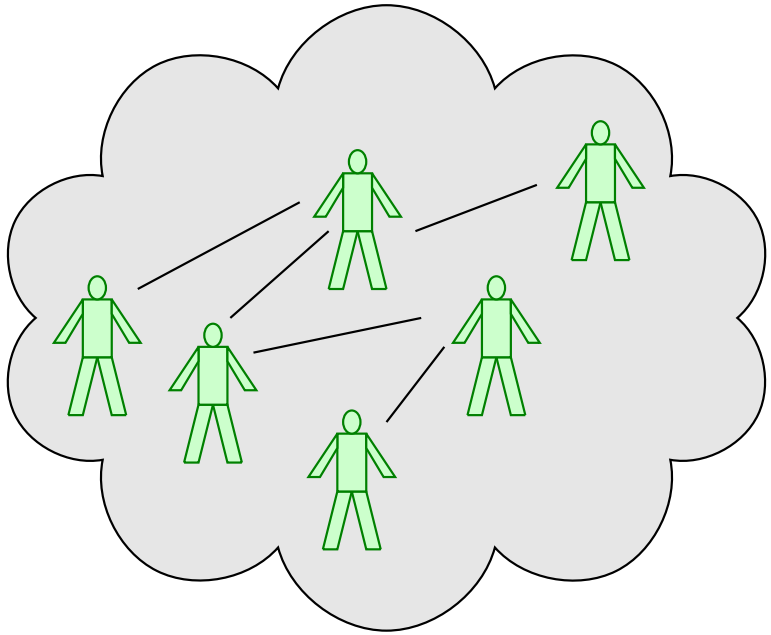
- SCEL is *parametric* wrt the actual language used to express policies.
- Currently we (Pugliese, Tiezzi) are defining a specific language based on XACML.
- When considering the operational semantics, we will see how policies are exploited to control components actions, their evolutions and their interactions.

SCEL: Syntax (in one slide)

ascens

SYSTEMS:	S	$::=$	$C \mid S_1 \parallel S_2 \mid (\nu n)S$
COMPONENTS:	C	$::=$	$\mathcal{I}[\mathcal{K}, \Pi, P]$
KNOWLEDGE:	K	$::=$	$\ldots$
PROCESSES:	P	$::=$	$\mathbf{nil} \mid a.P \mid P_1 + P_2 \mid P_1[P_2] \mid X \mid A(\bar{p}) \quad (A(\bar{f}) \triangleq P)$
ACTIONS:	a	$::=$	$\mathbf{get}(T)@c \mid \mathbf{qry}(T)@c \mid \mathbf{put}(t)@c \mid \mathbf{fresh}(n) \mid \mathbf{new}(\mathcal{I}, \mathcal{K}, \Pi, P)$
TARGETS:	c	$::=$	$n \mid x \mid \text{self}$
ITEMS:	t	$::=$	$\ldots$ – for the moment just tuples
TEMPLATES:	T	$::=$	$\ldots$ – for the moment tuples with variables

An ensemble



An ensemble is a set of components

- with the same goal and/or
- with compatible features and/or
- at the same locality and/or
- ...

Where are ensembles in SCEL?

ascens

- In the syntax, we just presented, there is no specific syntactic construct for building ensembles.
- We have experimented with different ways for modelling ensembles and their interaction.

Characterizing Ensembles

To identify those components that form an ensemble and guarantee general communication between members of the same ensemble we have considered:

1. Adding a specific syntactic category for ensembles
2. Enriching interfaces of some components with special attributes associated to components to single out groups of components forming an ensemble.
3. Using predicates to filter targets of send, retrieve and get operations.

ascens

Adding a specific syntactic category

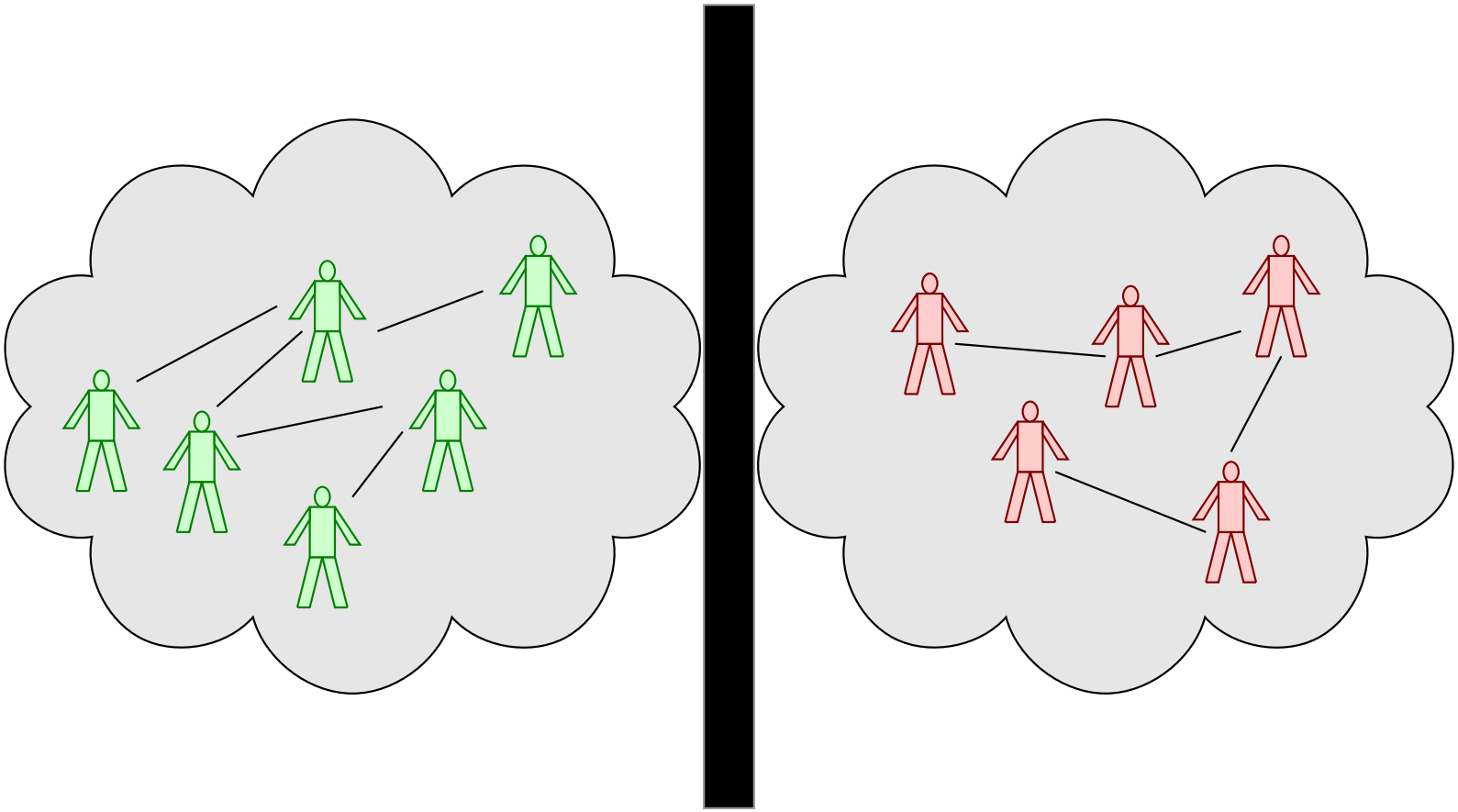
We explicitly declare the component that represents an ensemble, and whenever the target of an operation contains the name of an ensemble it will impact on all its components.

$$\textsc{Ensembles:} \quad C \quad ::= \quad \mathcal{I}[\mathcal{K}, \Pi, P]$$

One could then use the behavioural part P of the ensemble to distribute (retrieve) information to (from) the relevant partners and provide an ensemble-like (coordinated) behaviour.

This is the approach taken in process algebras with explicit localities or in programming language with distributed tuple space (e.g. Klaim).

Static ensembles



Drawback

Staticity of the aggregated structures. A component can be part of just one ensemble.

2. Attribute-based ensembles

ascens

There is no specific syntactic construct for building ensembles, they are dynamically formed by exploiting components interfaces and distinguished attributes like ensemble and membership. This is useful to:

- support flexibility in modeling ensemble forming, joining and leaving
- avoid structuring ensembles through rigid syntactic constructs
- control the communication capabilities of components.

Ensemble Interfaces

Interfaces specify (possibly dynamic) attributes (features) and functionalities (services provided). Each component C has in its interface attributes:

- *ensemble*: determines the actual components of the ensemble created and coordinated by C (n.b.: it might be *false*).
- *membership*: determines the ensembles which C is willing to be member of (n.b.: it might be *true*).

Components Attribute

ascens

ensemble attribute

- $\mathcal{I}.id \in \{n, m, p\}$
- $\mathcal{I}.active = yes \wedge \mathcal{I}.battery\_level > 30\%$
- $range_{max} > \sqrt{(\text{self}.x - \mathcal{I}.x)^2 + (\text{self}.y - \mathcal{I}.y)^2}$

membership attribute

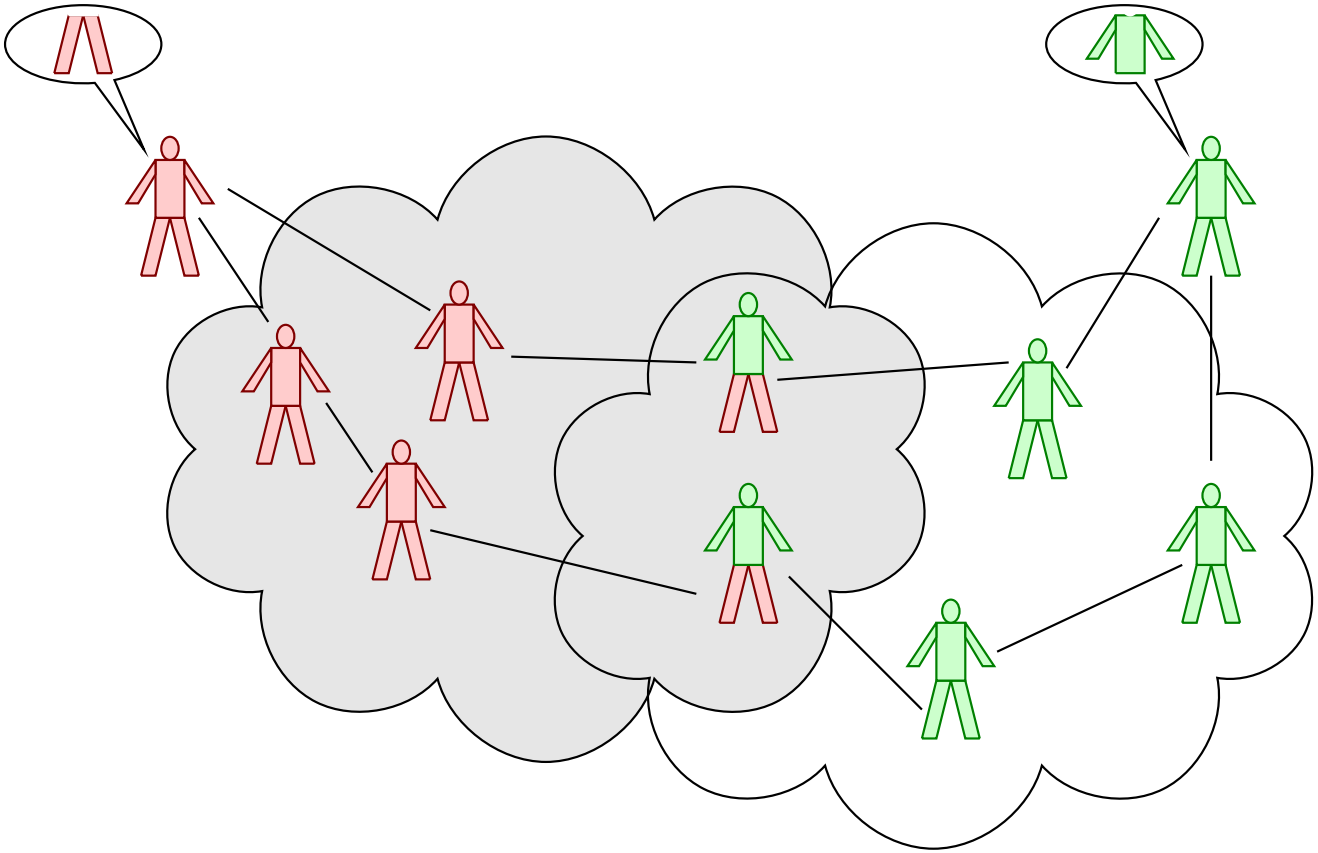
- *true*
- *false*
- $\mathcal{I}.trust\_level > medium$

Allowing ensemble as targets

By sending to, or retrieving and getting from super one components interacts with all the components of the same ensemble it is in.

$$\text{TARGETS:} \quad c \quad ::= \quad n \mid x \mid \text{self} \mid \text{super}$$

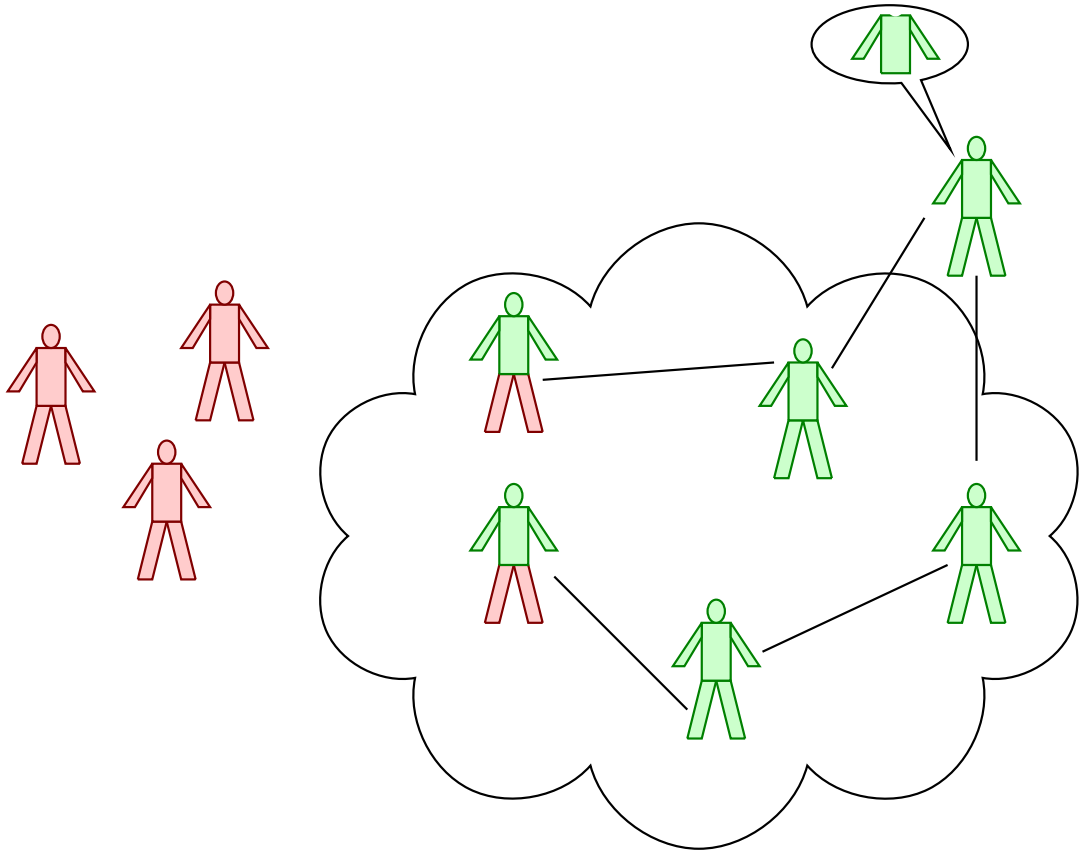
Attribute-based ensemble



Drawback

An ensemble dissolves if its coordinator disappears: single point of failure.

Attribute-based ensemble



Drawback

An ensemble dissolves if its coordinator disappears: single point of failure.

ascens

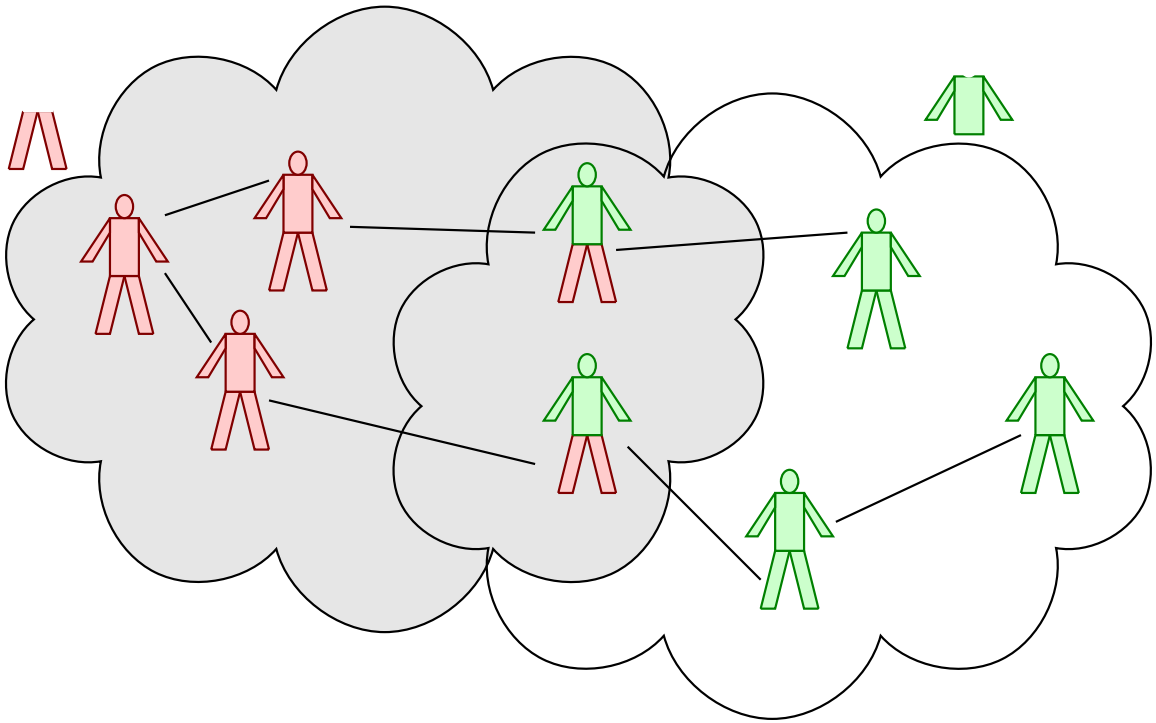
In order to guarantee the maximum degree of flexibility and to avoid critical component that could be not functional, we are currently investigating the possibility of predicate-based communication primitives that select the targets among those enjoying specific properties.

Allowing Predicates as targets

By sending to, or retrieving and getting from predicate P one components interacts with all the components that satisfy the same predicate.

$$\textsc{Targets:} \quad c \quad ::= \quad n \mid x \mid \text{self} \mid P$$

Predicate-based ensembles



Good point

No specific coordinator! Ensembles are determined by the predicates validated by each component.

SCEL: Operational Semantics

ascens

Structural operational semantics relies on the notion of Labelled Transition System (LTS)

LTS: a triple $\langle \mathcal{S}, \mathcal{L}, \rightarrow \rangle$

- A set of states $\mathcal{S}$
- A set of transition labels $\mathcal{L}$
- A labelled transition relation $\rightarrow \subseteq \mathcal{S} \times \mathcal{L} \times \mathcal{S}$ modelling the actions that can be performed from each state and the new state reached after each such transition

Semantics is structured in two layers:

1. Processes semantics specifies process commitments, i.e. the actions that processes can initially perform, while ignoring process allocation, available data, regulating policies, ...
2. Systems semantics, builds on process commitments and systems configuration to provide a full description of systems behavior.

Rules for Processes (excerpt)

$$a.P \downarrow_a P \qquad\qquad P \downarrow_\circ P$$

$$\frac{P \downarrow_\alpha P' \qquad Q \downarrow_\beta Q'}{P[\,Q\,] \downarrow_{\alpha[\,\beta\,]} P'[\,Q'\,]}$$

- $a.P$ executes action a and then behaving like process P
- $\downarrow_\circ$ indicates that process P may always decide to stay idle
- The semantics of $P[\,Q\,]$ at process level is very permissive and generates all combinations of the commitments of the involved processes; its behaviour is refined at systems level when policies enter the game.

ascens

From process actions to component actions

$$\frac{P \downarrow_{\alpha} P' \qquad \Pi, \mathcal{I} : \alpha \succ \lambda, \sigma, \Pi'}{\mathcal{I}[\mathcal{K}, \Pi, P] \xrightarrow{\lambda} \mathcal{I}[\mathcal{K}, \Pi', P'\{\sigma\}]}$$

ascens

From process actions to component actions

$$\frac{P \downarrow_{\alpha} P' \qquad \Pi, \mathcal{I} : \alpha \succ \lambda, \sigma, \Pi'}{\mathcal{I}[\mathcal{K}, \Pi, P] \xrightarrow{\lambda} \mathcal{I}[\mathcal{K}, \Pi', P'\{\sigma\}]}$$

Intra-component withdrawal

$$\frac{\mathcal{I}[\mathcal{K}, \Pi, P] \xrightarrow{\mathcal{I}:t \triangleleft n} \mathcal{I}[\mathcal{K}, \Pi, P'] \quad n = \mathcal{I}.id \quad \mathcal{K} \ominus t = \mathcal{K}' \quad \Pi, \mathcal{I} \vdash \mathcal{I} : t \bar{\triangleleft} \mathcal{I}, \Pi'}{\mathcal{I}[\mathcal{K}, \Pi, P] \xrightarrow{\tau} \mathcal{I}[\mathcal{K}', \Pi', P']}$$

SOS Rules for Systems (excerpt)

ascens

From process actions to component actions

$$\frac{P \downarrow_{\alpha} P' \qquad \Pi, \mathcal{I} : \alpha \succ \lambda, \sigma, \Pi'}{\mathcal{I}[\mathcal{K}, \Pi, P] \xrightarrow{\lambda} \mathcal{I}[\mathcal{K}, \Pi', P'\{\sigma\}]}$$

Intra-component withdrawal

$$\frac{\mathcal{I}[\mathcal{K}, \Pi, P] \xrightarrow{\mathcal{I}:t \lhd n} \mathcal{I}[\mathcal{K}, \Pi, P'] \quad n = \mathcal{I}.id \quad \mathcal{K} \ominus t = \mathcal{K}' \quad \Pi, \mathcal{I} \vdash \mathcal{I} : t \bar{\lhd} \mathcal{I}, \Pi'}{\mathcal{I}[\mathcal{K}, \Pi, P] \xrightarrow{\tau} \mathcal{I}[\mathcal{K}', \Pi', P']}$$

Inter-component, intra-ensemble withdrawal

$$\frac{S_1 \xrightarrow{\mathcal{I}:t \lhd n} S_1' \qquad S_2 \xrightarrow{\mathcal{I}:t \bar{\lhd} \mathcal{J}} S_2' \qquad \mathcal{J}.id = n}{S_1 \parallel S_2 \xrightarrow{\tau} S_1' \parallel S_2'}$$

ascens

Inter-component, intra-ensemble withdrawal

$$\frac{S_1 \xrightarrow{\mathcal{I}:t \triangleleft \mathsf{P}} S_1' \qquad S_2 \xrightarrow{\mathcal{I}:t \bar{\triangleleft} \mathcal{J}} S_2' \qquad \mathcal{J} \models \mathsf{P}}{S_1 \parallel S_2 \xrightarrow{\tau} S_1' \parallel S_2'}$$

Interaction Predicates: The interleaving case

$$\frac{\mathcal{E}[\![\ T\ ]\!]\mathcal{I} = T' \qquad \mathcal{N}[\![\ c\ ]\!]_{\mathcal{I}} = c' \qquad \mathit{match}(T', t) = \sigma}{\Pi_\oplus, \mathcal{I} : \mathbf{get}(T)@c \succ \mathcal{I} : t \triangleleft c', \sigma, \Pi_\oplus}$$

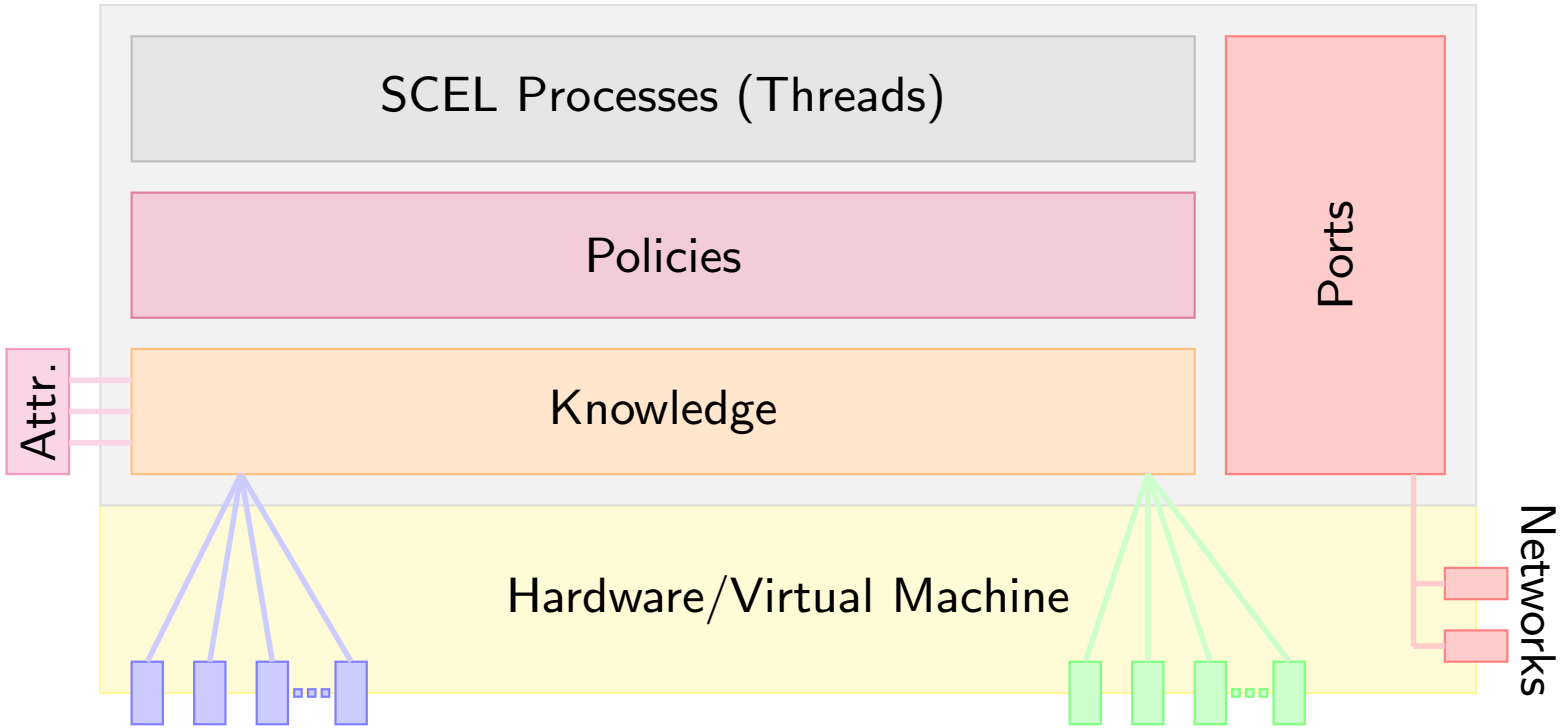
N.B: c' can be a component identifier or a predicate.

$$\frac{\Pi_\oplus, \mathcal{I} : \alpha \succ \lambda, \sigma, \Pi_\oplus}{\Pi_\oplus, \mathcal{I} : \alpha[\circ] \succ \lambda, \sigma, \Pi_\oplus} \qquad \frac{\Pi_\oplus, \mathcal{I} : \alpha \succ \lambda, \sigma, \Pi_\oplus}{\Pi_\oplus, \mathcal{I} : \circ[\alpha] \succ \lambda, \sigma, \Pi_\oplus}$$

Basic design principles...

1. no centralized control
2. heavy use of *recurrent patterns* to simply the development of specific
   - knowledge
     - a single interface that contains basic methods to interact with knowledge
   - policies
     - based on the pattern *composite* (policies are structured as a stack)
   - ...
3. use of *open technologies* to support the integration with other tools/frameworks or with alternative implementations of SCEL

SCEL component



Input devices/Sensors
(GPS, Temperature, Battery level,CPU load. . .)

Output devices/Actuators

Ongoing & Future Work

ascens

We have concentrated on modelling behaviors of components and their interactions; taking into account spatial distribution. We are currently assessing this work and tackling other research items.
We are :

- working on interaction policies to study the possibility of modelling different for of synchronization and communication
- considering different knowledge repositories and ways of expressing goals by analyzing different knowledge representation languages
- assessing the impact and the sensitivity of different adaptation patterns.
- developping quantitative variants of SCEL to support components in taking decisions (e.g. via probabilistic model checking).

Many thanks for your time