Software Engineering and Service-Oriented Systems

– A formal approach to autonomic systems programming –

Francesco Tiezzi

IMT - Institutions, Markets, Technologies

Institute for Advanced Studies Lucca

Lucca, Italy - September, 2013

In co-operation with ASCENS members, in particular R. De Nicola, G. Ferrari, M. Loreti and R. Pugliese

Outline

1. Autonomic Computing and Ensembles Programming
2. Programming Abstractions for Autonomic Computing
3. Ensembles Modelling in SCEL
4. Operational Semantics
5. A Simple Access Control Policy Language
6. A Run-time Environment for SCEL Programs

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 of 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

A possible answer to the challenges posed by ensemble systems is:

Autonomic Computing

A possible answer to the challenges posed by ensemble systems is:

Autonomic Computing

Systems can manage themselves by continuously

- monitoring their behaviour (self-awareness) and their working environment (context-awareness)
- analysing the acquired knowledge to identify relevant changes
- planning reconfigurations in order to meet their requirements
- executing plan actions

A possible answer to the challenges posed by ensemble systems is:

Autonomic Computing

Systems can manage themselves by continuously

- monitoring their behaviour (self-awareness) and their working environment (context-awareness)
- analysing the acquired knowledge to identify relevant changes
- planning reconfigurations in order to meet their requirements
- executing plan actions

Self-management requires guaranteeing the so-called **self-* properties**:

- self-configuration
- self-healing
- self-optimization
- self-protection

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

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

Interfaces and Attributes

- SCs are entities with dedicated knowledge units and resources
- Each SC is equipped with an interface, consisting of a collection of attributes, describing component's features such as identity, spatial coordinates, group memberships, trust level, response time, etc.

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

Interfaces and Attributes

- SCs are entities with dedicated knowledge units and resources
- Each SC is equipped with an interface, consisting of a collection of attributes, describing component's features such as identity, spatial coordinates, group memberships, trust level, response time, etc.

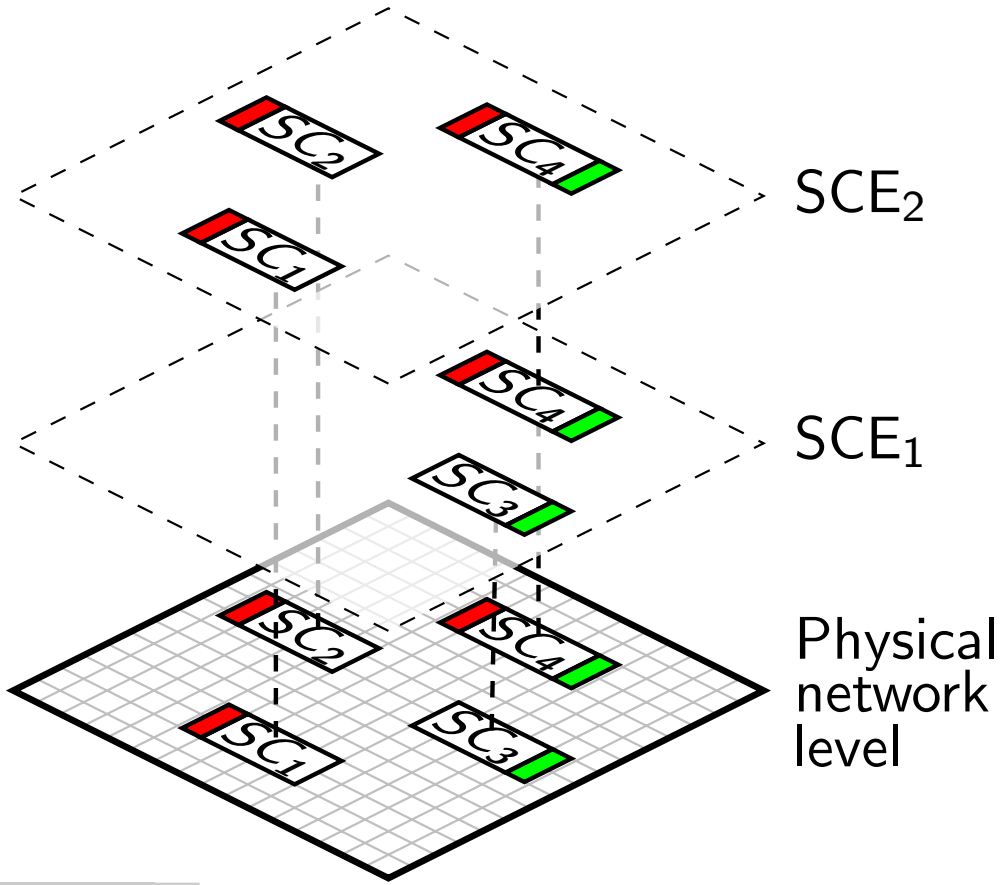
Attribute-based communication

- SCs use attributes to dynamically organize themselves into SCEs
- Predicates over SCs' attributes specify the targets of communication actions, i.e. the members of an ensembles

ascens

Service-Component Ensembles

SCEs can be thought of as logical layers (built on top of the physical SCs network) that identify dynamic (overlay) subnetworks of SCs



ascens

We aim 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

Service-Component Ensemble Language (SCEL)

A kernel language designed with

- programming abstractions necessary for
  - directly representing Knowledge, Behaviors and Aggregations according to specific Policies
  - naturally programming interaction, adaptation and self- and context-awareness (Autonomic Computing)
- 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

SCEL provides a common semantic framework for describing the meaning of these abstractions and their interplay, while minimizing overlaps and incompatibilities

- SCEL syntax fully specifies only constructs for modeling Behaviors and Aggregations and is parametric with respect to Knowledge and Policies
- This choice permits integrating different approaches to knowledge handling or to policies specifications within SCEL and to easily superimpose ACEs on top of heterogeneous ACs
- SCEL is thought of as a kernel language based on which different full-blown languages can be designed

SCEL is a blending of different concepts that have emerged in different fields of Computer Science and Engineering

- **Software engineering**: separation of concerns and importance of component-based design
- **Multi-agent systems**: relevance of knowledge handling and of spatial representation
- **Middleware and network architectures**: importance of flexibility in communication
- **Distributed systems' security**: the role of policies
- **Process algebras**: importance of minimality and formality

Contribution

New language with appropriate programming abstractions for autonomic computing and their reconciliation under a single roof with a uniform formal semantics

Programming Abstractions for Autonomic Computing

SCEL: programming abstractions

ascens

SCEL provides primitives and constructs for dealing with 4 programming abstractions

1. Knowledge: to describe how data, information and knowledge is manipulated and shared
2. Behaviours: to describe how systems of components progress
3. Aggregations: to describe how different entities are brought together to form SCs and to construct the software architecture of SCEs
4. Policies: to control and adapt the actions of the different SCs

1. Knowledge

ascens

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

Knowledge representation: items stored in repositories

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

Knowledge handling mechanisms

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

1. Knowledge (and Awareness)

ascens

To guarantee adaptivity, some kinds of knowledge items and handling mechanisms are required

Application data

Used for the progress of the computation

Control data

Providing information about:

- SC's current status (e.g. its position, its battery level): self-awareness
- the environment (e.g. data from sensors): context-awareness

Knowledge handling mechanisms

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

2. Behaviors

ascens

Behaviours are modelled as **processes** in the style of process calculi

Interaction

Obtained by allowing processes to access knowledge repositories, possibly of other components

Adaptation

Modelled by retrieving from the knowledge repositories

- information about the changing environment and the components status
- the code to execute for reacting to these changes

Self-configuration, self-healing and self-optimization properties are expressed by exploiting SCEL's interaction features

ascens

Processes are the SCEL's active computational units

Processes

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

Processes are built up from the inert process **nil** via

- *action prefixing*: $a.P$
- *nondeterministic choice*: $P_1 + P_2$
- *controlled composition*: $P_1[\,P_2\,]$
- *process variable*: X
- *parameterized process invocation*: $A(\bar{p})$
- *parameterised process definition*: $A(\bar{f}) \triangleq P$

ascens

Processes are the SCEL's active computational units

Processes

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

Processes are built up from the inert process **nil** via

- *action prefixing*: $a.P$
- *nondeterministic choice*: $P_1 + P_2$
- *controlled composition*: $P_1[\,P_2\,]$
- *process variable*: X
- *parameterized process invocation*: $A(\bar{p})$
- *parameterised process definition*: $A(\bar{f}) \triangleq P$

$P_1[\,P_2\,]$ can be seen as a generalization of the many "parallel compositions" of process calculi

ascens

Processes are the SCEL's active computational units

Processes

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

Processes are built up from the inert process **nil** via

- *action prefixing*: $a.P$
- *nondeterministic choice*: $P_1 + P_2$
- *controlled composition*: $P_1[\,P_2\,]$
- *process variable*: X
- *parameterized process invocation*: $A(\bar{p})$
- *parameterised process definition*: $A(\bar{f}) \triangleq P$

Variables X support higher-order communication, namely the capability to exchange (the code of) a process and possibly execute it

ascens

Processes are the SCEL's active computational units

Processes

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

Processes are built up from the inert process **nil** via

- *action prefixing*: $a.P$
- *nondeterministic choice*: $P_1 + P_2$
- *controlled composition*: $P_1[\,P_2\,]$
- *process variable*: X
- *parameterized process invocation*: $A(\bar{p})$
- *parameterised process definition*: $A(\bar{f}) \triangleq P$

2. Behaviours (and Actions)

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

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

Rôle of Actions

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

  t is a knowledge item, T is a template to select knowledge items
- create new names - $\mathbf{fresh}(n)$ |
- create 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 to

- model resource *allocation* and *distribution*
- reflect the idea of *administrative domains*, i.e. the authority controlling a given set of resources and computing agents

Modelled by resorting to the notions of system, component and ensemble

ascens

Aggregations

Describe how different entities are brought together to

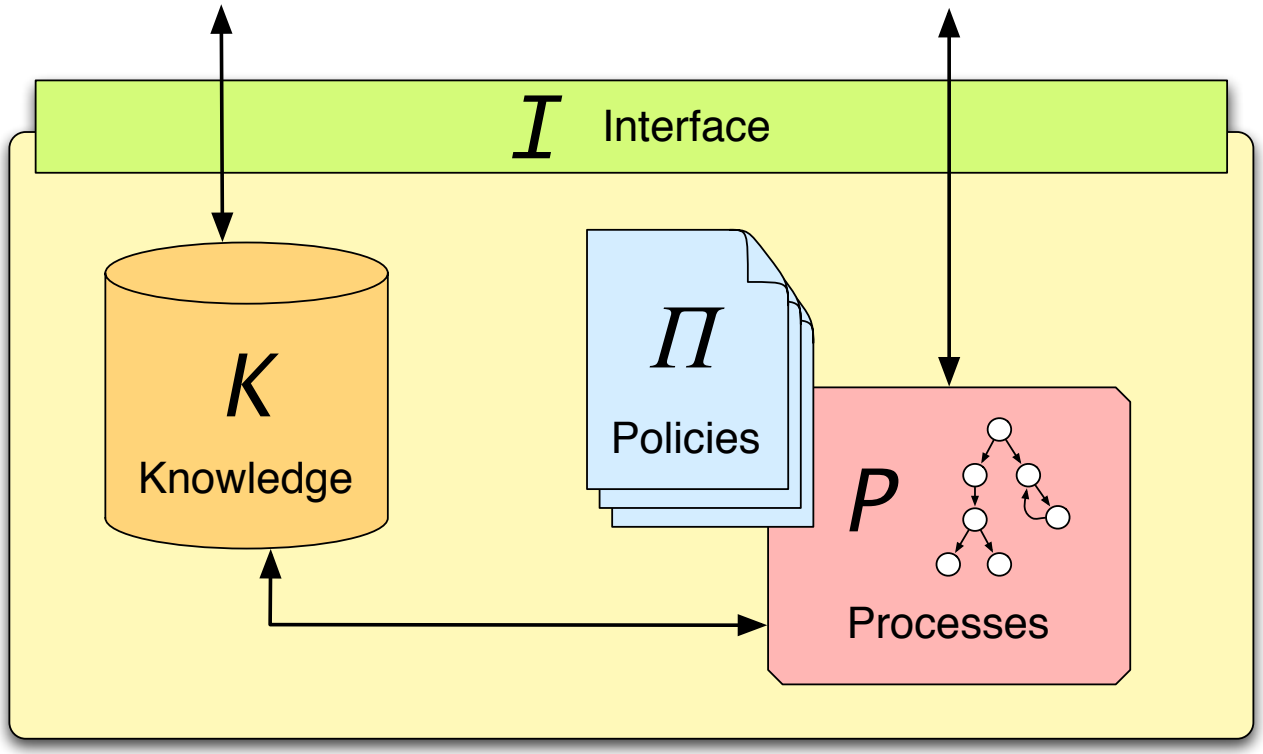
- model resource *allocation* and *distribution*
- reflect the idea of *administrative domains*, i.e. the authority controlling a given set of resources and computing agents

Modelled by resorting to the notions of system, component and ensemble

Systems

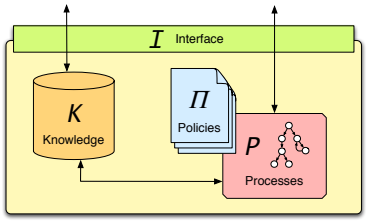
$$S \quad ::= \quad C \quad | \quad S_1 \parallel S_2 \quad | \quad (\nu n)S$$

- 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



Components

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



Components

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

- Interface $\mathcal{I}$: containing information about the component itself in the form of **attributes**, each component C has attribute:
  - *id*: referring to the name of the component C
- Knowledge repository $\mathcal{K}$: providing application and control data together with specific knowledge handling mechanisms
- Policies Π: regulating inter-component and intra-component interactions
- Process P: performing the local computation, coordinating their interaction with the knowledge repository and dealing with adaptation

4. Policies

ascens

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

- Interaction: interaction predicates, ...
- Resource usage: SLA, accounting, ...
- Security: access control, trust, reputation, ...
- System configuration: adaptation, ...

- SCEL is *parametric* wrt the actual language used to express policies
  - we consider here SACPL, a simple language for access control
- When considering the operational semantics, we will see how policies are exploited to control components actions, their evolutions and their interactions

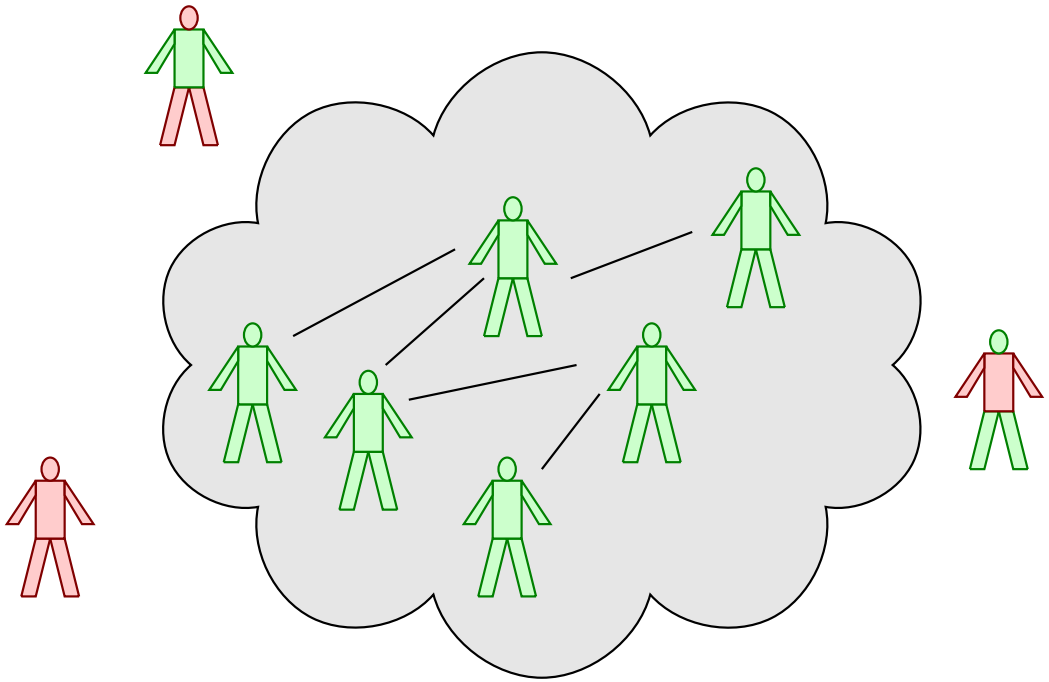
SCEL: Syntax (in one slide)

ascens

SYSTEMS:	S	$::=$	$C \mid S_1 \parallel S_2 \mid (\nu n)S$
COMPONENTS:	C	$::=$	$\mathcal{I}[\mathcal{K}, \Pi, P]$
KNOWLEDGE:	K	$::=$	$\ldots$ – *currently, just tuple spaces*
POLICIES:	Π	$::=$	$\ldots$ – *interaction policies and* SACPL *terms*
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	$::=$	$n \mid x \mid \mathbf{self} \mid \ldots$ – *ensembles*
ITEMS:	t	$::=$	$\ldots$ – *tuples*
TEMPLATES:	T	$::=$	$\ldots$ – *tuples with variables*

Ensembles Modelling in SCEL

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
- Different ways for modelling ensembles and their interaction have been experimented

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 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: ensembles as components

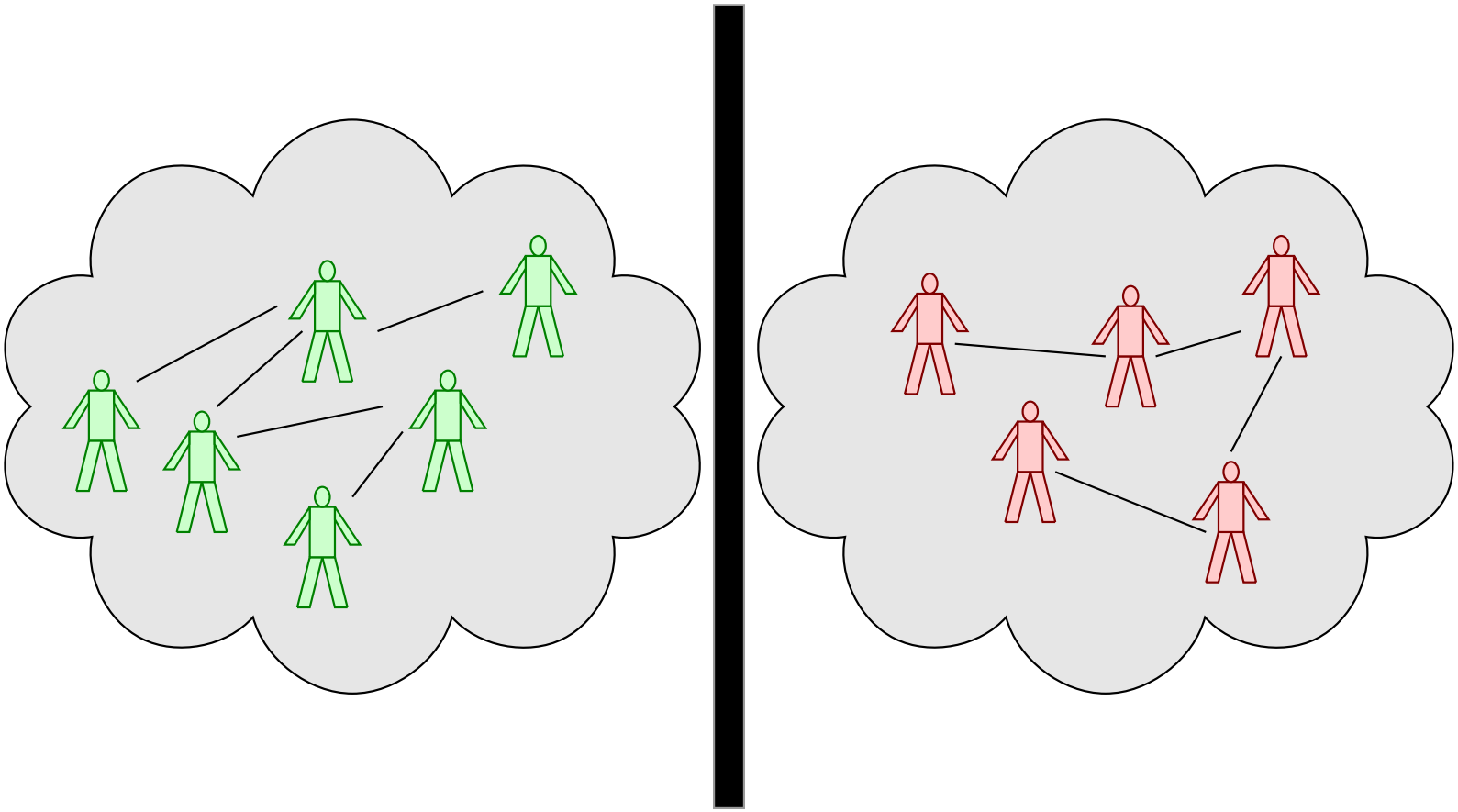
We explicitly declare the component that represents an ensemble, and whenever the target of an operation contains the name of an ensemble it will impact on all its components

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

- The behavioural part P of the ensemble could distribute (retrieve) information to (from) the relevant partners and provide an ensemble-like (coordinated) behaviour
- Components could be nested

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. 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

- $id \in \{n, m, p\}$
- $active = \text{yes} \wedge battery\_level > 30\%$

membership attribute

- true, false
- $trust\_level > \text{medium}$

Components Attribute

ascens

ensemble attribute

- $id \in \{n, m, p\}$
- $active = \text{yes} \wedge battery\_level > 30\%$

membership attribute

- true, false
- $trust\_level > \text{medium}$

Components interactions

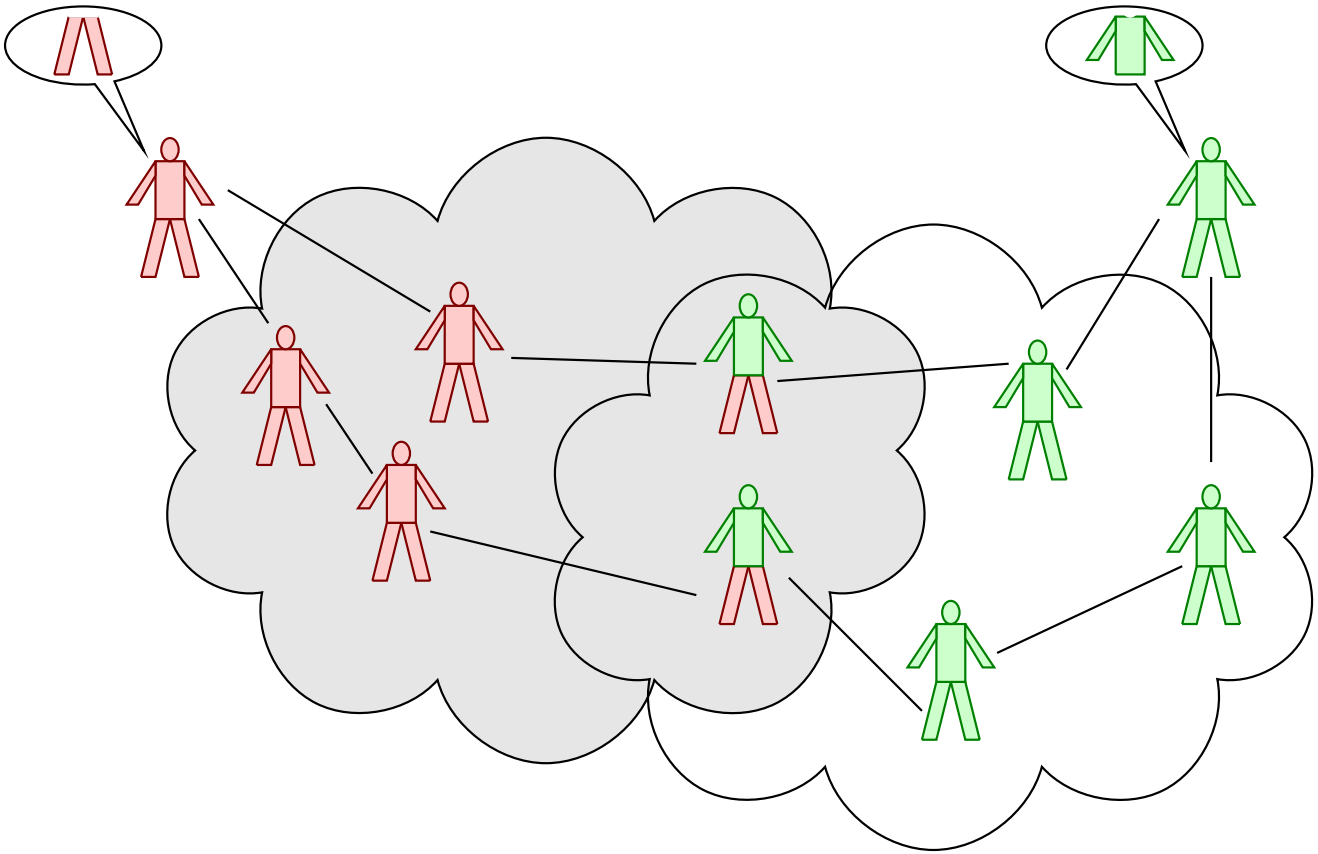
Only components within the same ensemble are allowed to interact

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

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

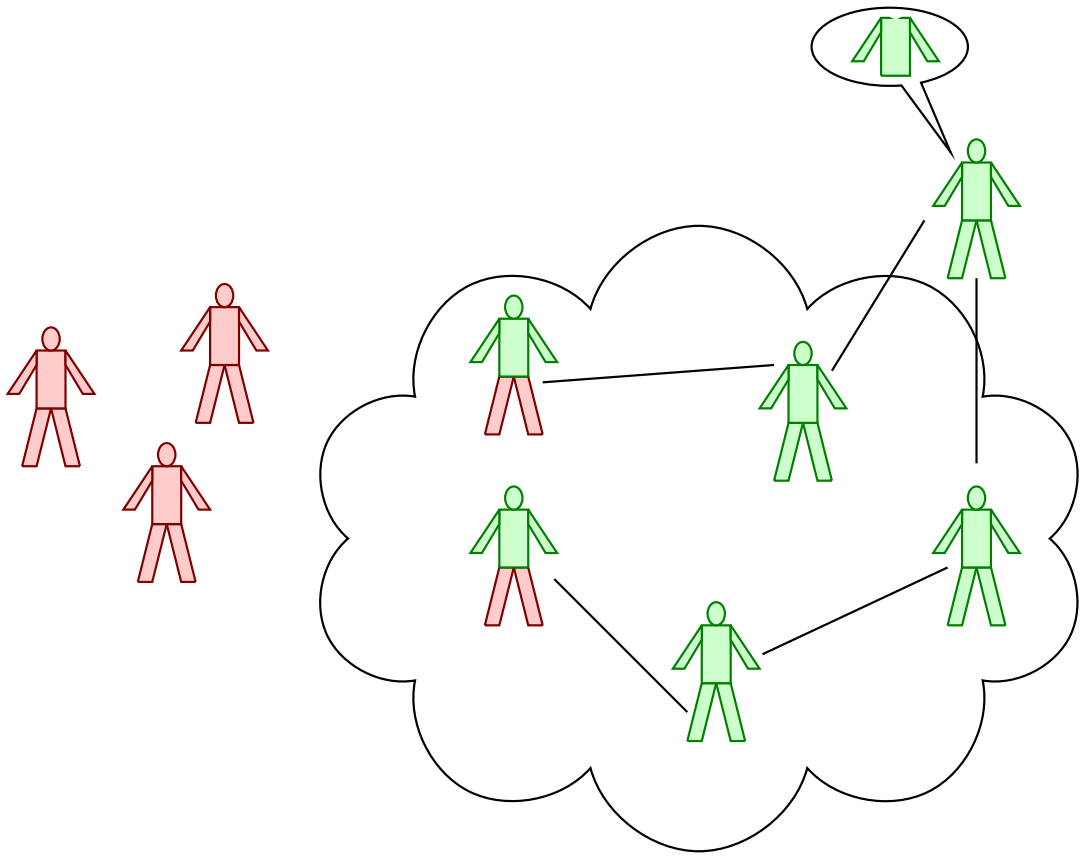
Attribute-based ensemble



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

3. Ensembles as predicates

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: ensembles as predicates

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

$$\textsc{Targets:} \quad c \quad ::= \quad n \mid x \mid \textbf{self} \mid \mathrm{P} \mid \mathrm{p}$$

Predicates

Boolean-valued expressions obtained by logically combining the evaluation of relations between attributes and expressions

3. Ensembles as predicates

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: ensembles as predicates

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

$$\textsc{Targets:} \quad c \quad ::= \quad n \mid x \mid \textbf{self} \mid \mathrm{P} \mid \mathrm{p}$$

Predicates

- $\mathit{active} = \mathrm{yes} \ \wedge \ \mathit{battery\_level} > 30\%$
- $\textbf{this}.\mathit{status} = \mathrm{sending} \ \wedge \ \mathit{status} = \mathrm{receiving}$

3. Ensembles as predicates

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: ensembles as predicates

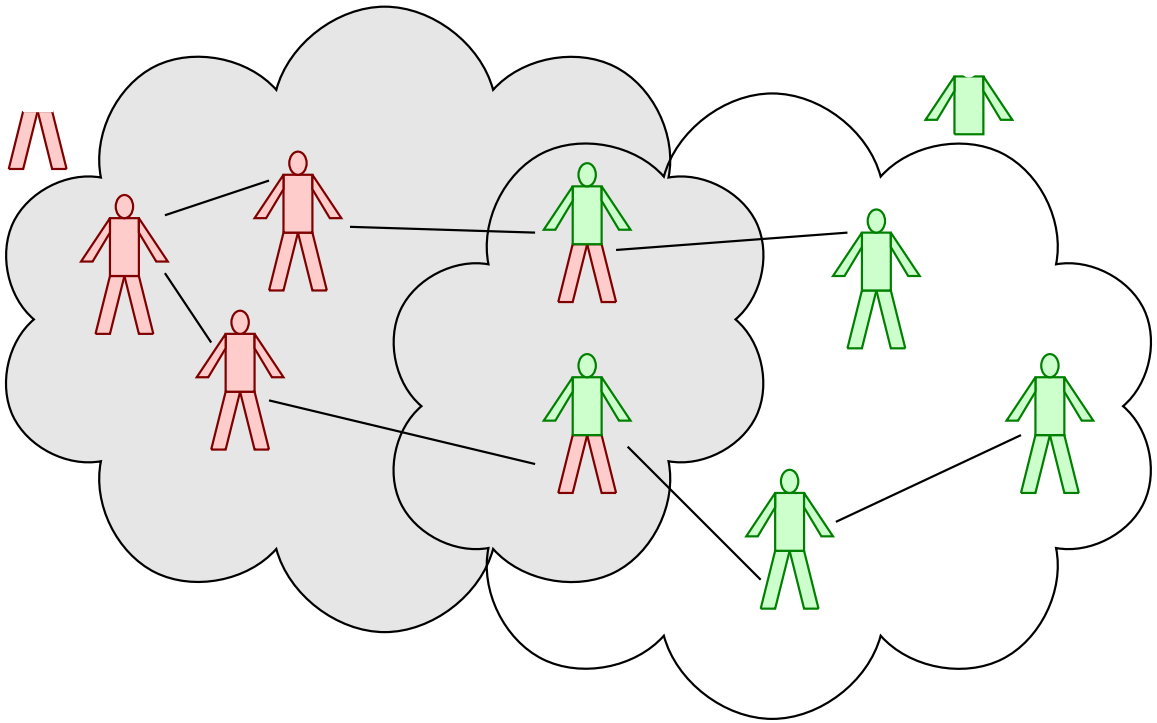
By sending to, or retrieving and getting from predicate P one components interacts with all the components that satisfy the same predicate

$$\textsc{Targets:} \quad c \quad ::= \quad n \mid x \mid \textbf{self} \mid \mathrm{P} \mid \mathrm{p}$$

Predicates

Name p, exposed as an interface attribute, is a reference to a predicate stored in the component's repository; it enables changing predicates

Predicate-based ensembles



Good point

No specific coordinator!

Ensembles are determined by the predicates validated by each component

An simple example

ascens

A swarm robotics scenario

- Robots of a swarm have to reach different zones according to the tasks that they have to do (here, $task_1$ or $tasks_2$)
  - e.g. help other robots, reach a safe area, clear a minefield, etc.
- Robots are not informed about the position of the two target zones
  - each robot follows a random walk
  - when a robot reaches its target area, it *publishes* the location within its local repository to make it available to robots with the same task
  - informed robots can then move directly towards the target, by saving time wrt random walking (i.e., they **self-optimise** their behaviour)

An simple example

ascens

A swarm robotics scenario

- Robots of a swarm have to reach different zones according to the tasks that they have to do (here, $task_1$ or $tasks_2$)
  - e.g. help other robots, reach a safe area, clear a minefield, etc.
- Robots are not informed about the position of the two target zones
  - each robot follows a random walk
  - when a robot reaches its target area, it *publishes* the location within its local repository to make it available to robots with the same task
  - informed robots can then move directly towards the target, by saving time wrt random walking (i.e., they **self-optimise** their behaviour)

The robotics scenario can be expressed in SCEL as a system

$$S \triangleq \mathcal{I}_1[\mathcal{K}_1, \Pi_1, P_1] \parallel \mathcal{I}_2[\mathcal{K}_2, \Pi_2, P_2] \parallel \mathcal{I}_3[\mathcal{K}_3, \Pi_3, P_3] \parallel \mathcal{I}_4[\mathcal{K}_4, \Pi_4, P_4] \parallel \ldots$$

An simple example

- The first robot is the component $\mathcal{I}_1[\mathcal{K}_1, \Pi_1, P_1]$
- The robot runs process P_1 defined as

$$\begin{array}{l} \textbf{qry}(\text{“}\mathit{targetLocation}\text{”}, ?x, ?y)@(\mathit{task} = \text{“}\mathit{task}_1\text{”}) \ . \\ \textbf{put}(\text{“}\mathit{targetLocation}\text{”}, x, y)@\textbf{self} \ . \\ P_1' \end{array}$$

Symbol ‘?’ indicates a variable binder in a template

Operational Semantics

ascens

Structural operational semantics relies on the notion of LTS

Labelled Transition System (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

SCEL: Operational Semantics

ascens

Structural operational semantics relies on the notion of LTS

Labelled Transition System (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

SCEL's 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

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

$$\frac{P \downarrow_\alpha P'}{P + Q \downarrow_\alpha P'} \qquad \frac{Q \downarrow_\alpha Q'}{P + Q \downarrow_\alpha Q'} \qquad \frac{P\{\bar{p}/\bar{f}\} \downarrow_\alpha P'}{A(\bar{p}) \downarrow_\alpha P'} \; A(\bar{f}) \triangleq P$$

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

- $a.P$ 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

SOS Rules for Systems (excerpt)

From process actions to component actions

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

SOS Rules for Systems (excerpt)

ascens

From process actions to component actions

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

Interaction Predicates: The interleaving case

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

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

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

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

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

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

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

Authorization predicates

We will see an example later on ...

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

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

Inter-component, point-to-point withdrawal

$$\frac{S_1 \xrightarrow{\mathcal{I}:t \lhd n} S_1' \qquad S_2 \xrightarrow{\mathcal{I}:t \bar{\lhd} \mathcal{J}} S_2' \qquad \mathcal{J}.id = n \qquad \mathcal{I}.\pi \vdash \mathcal{I} : t \bar{\lhd} \mathcal{J}, \Pi'}{S_1 \parallel S_2 \xrightarrow{\tau} S_1'[\mathcal{I}.\pi := \Pi'] \parallel S_2'}$$

ascens

Inter-component, group-oriented withdrawal

$$\frac{S_1 \xrightarrow{\mathcal{I}:t \lhd \mathsf{P}} S_1' \qquad S_2 \xrightarrow{\mathcal{I}:t \bar{\lhd} \mathcal{J}} S_2' \qquad \mathcal{J} \models \mathsf{P} \qquad \mathcal{I}.\pi \vdash \mathcal{I} : t \bar{\lhd} \mathcal{J}, \Pi'}{S_1 \parallel S_2 \xrightarrow{\tau} S_1'[\mathcal{I}.\pi := \Pi'] \parallel S_2'}$$

Simple Access Control Policy Language

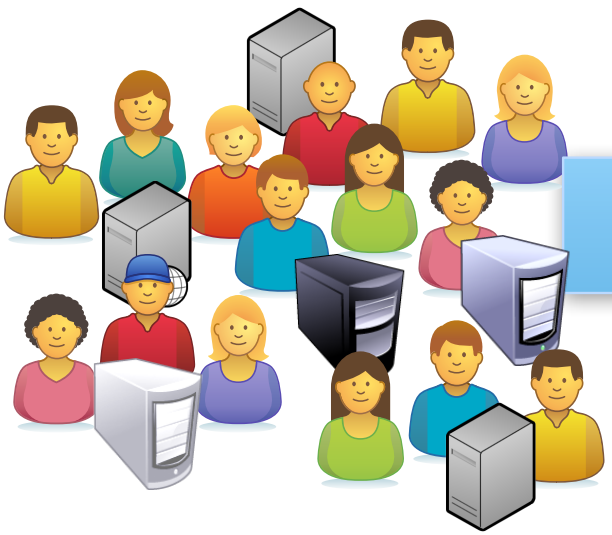
SACPL: Simple Access Control Policy Language

- A language for defining access control policies and access requests
- Follows the PBAC model
- Inspired to, but much simpler than, XACML
- Integrated with SCEL

SACPL: Simple Access Control Policy Language

- A language for defining access control policies and access requests
- Follows the PBAC model
- Inspired to, but much simpler than, XACML
- Integrated with SCEL

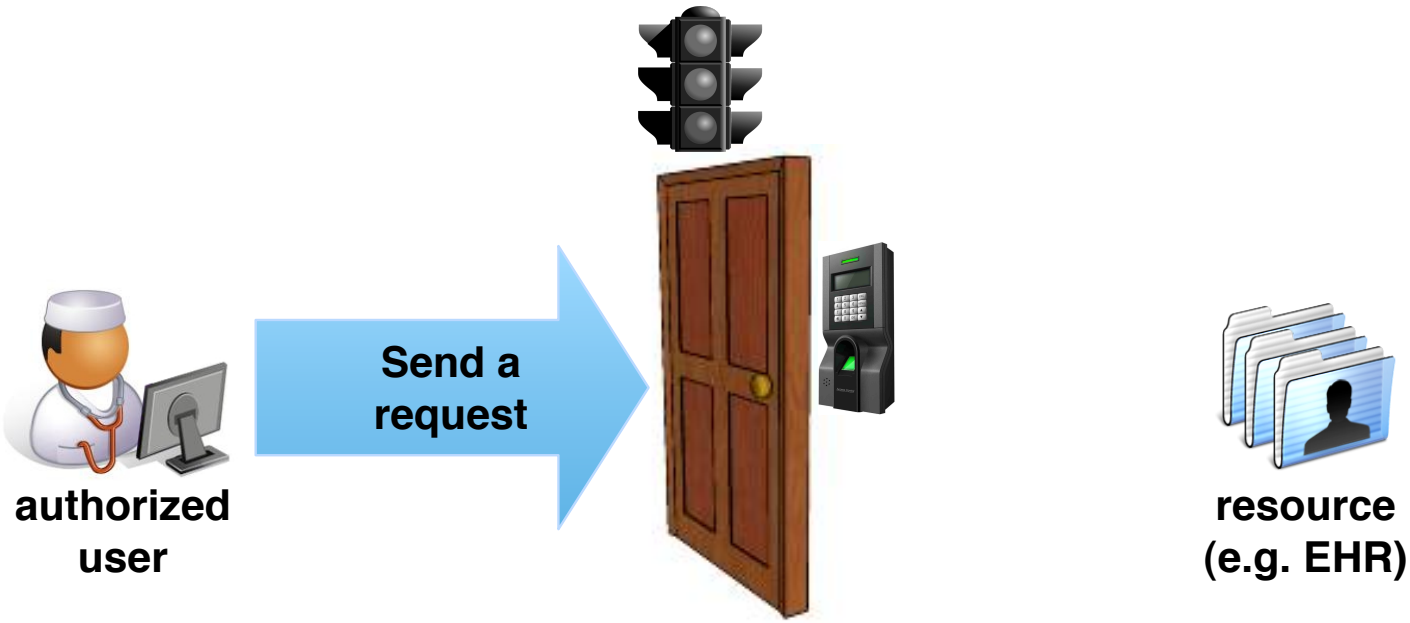
Access Control

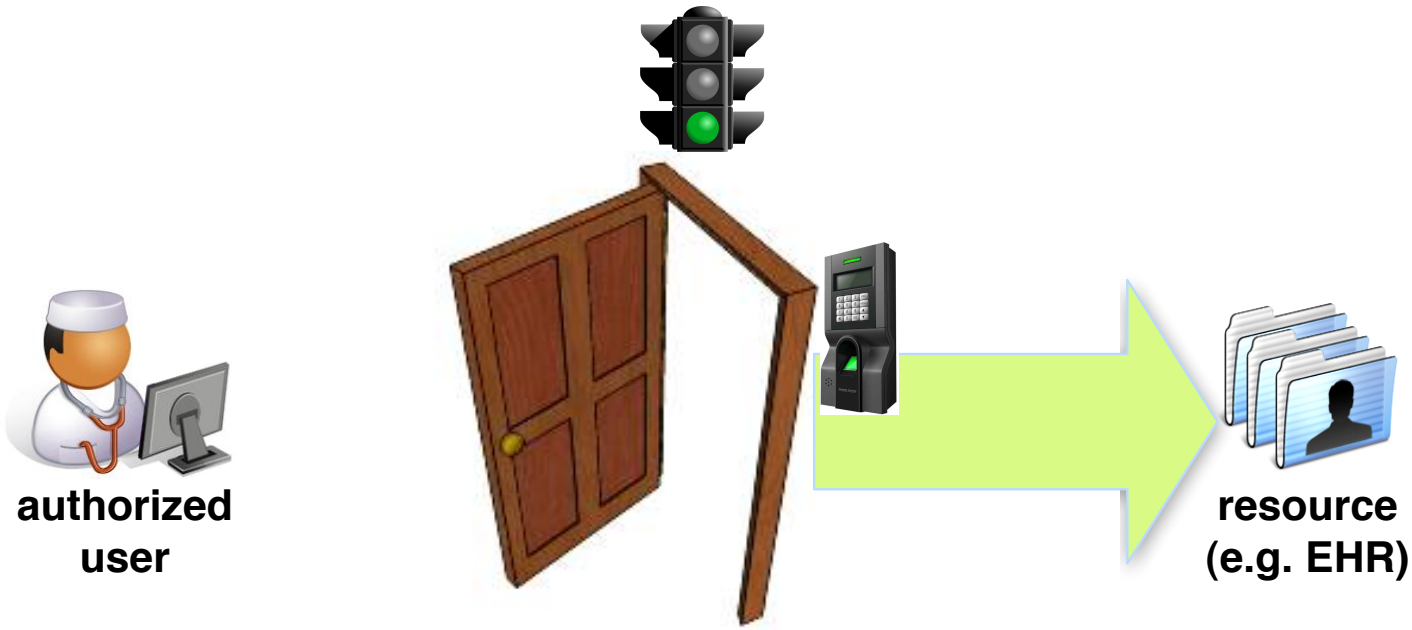


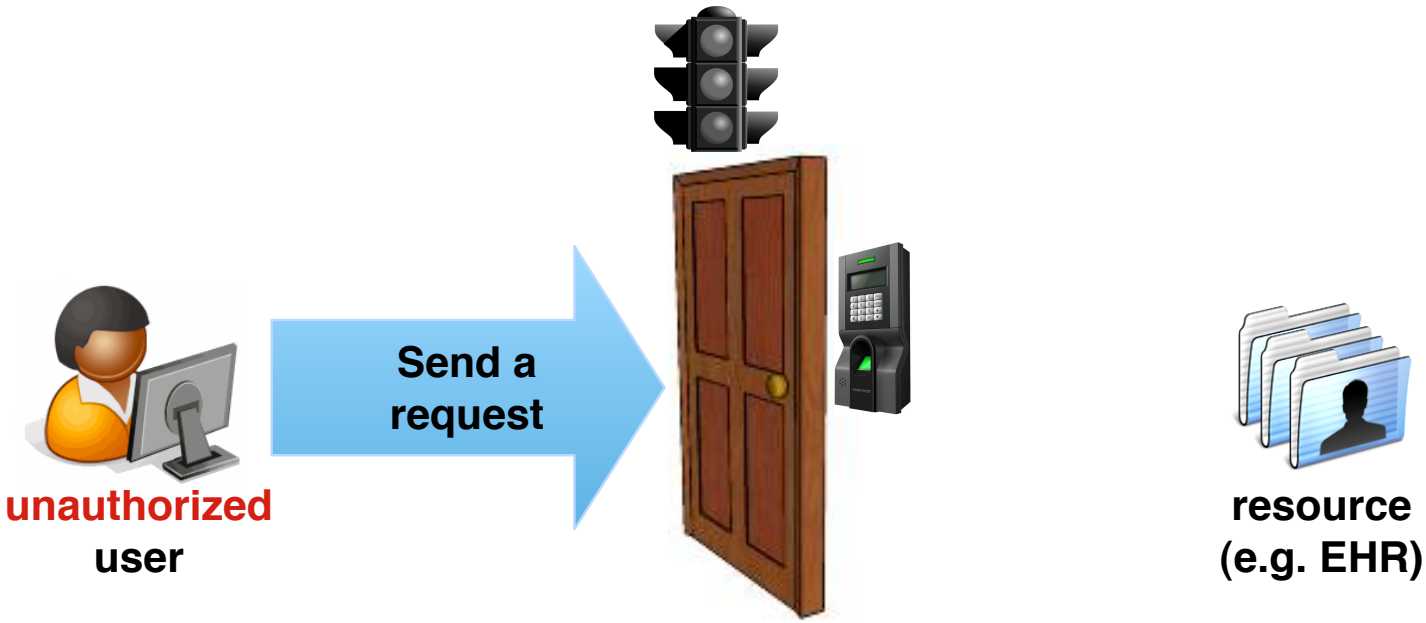
Require access to resources and systems

Authorization service
(access control mechanism)

Access Control

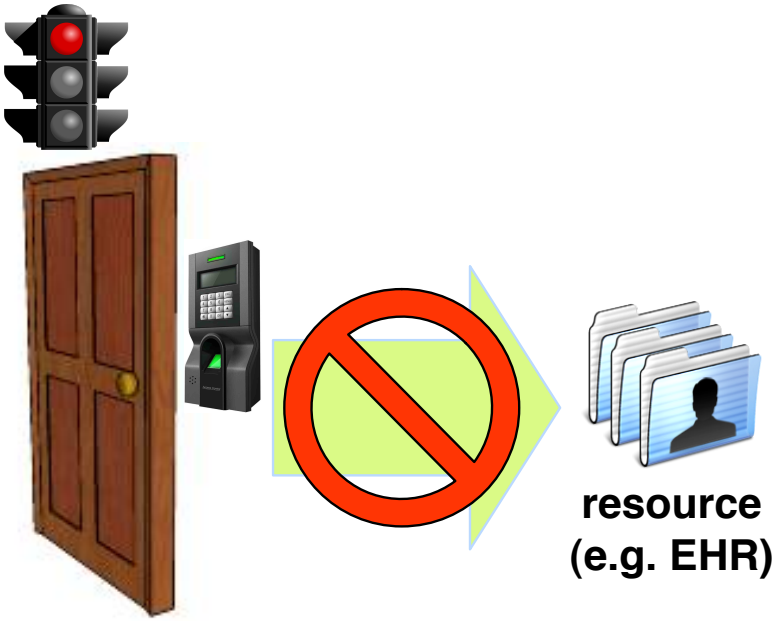



authorized
user
resource
(e.g. EHR)


Send a
request
unauthorized
user
resource
(e.g. EHR)

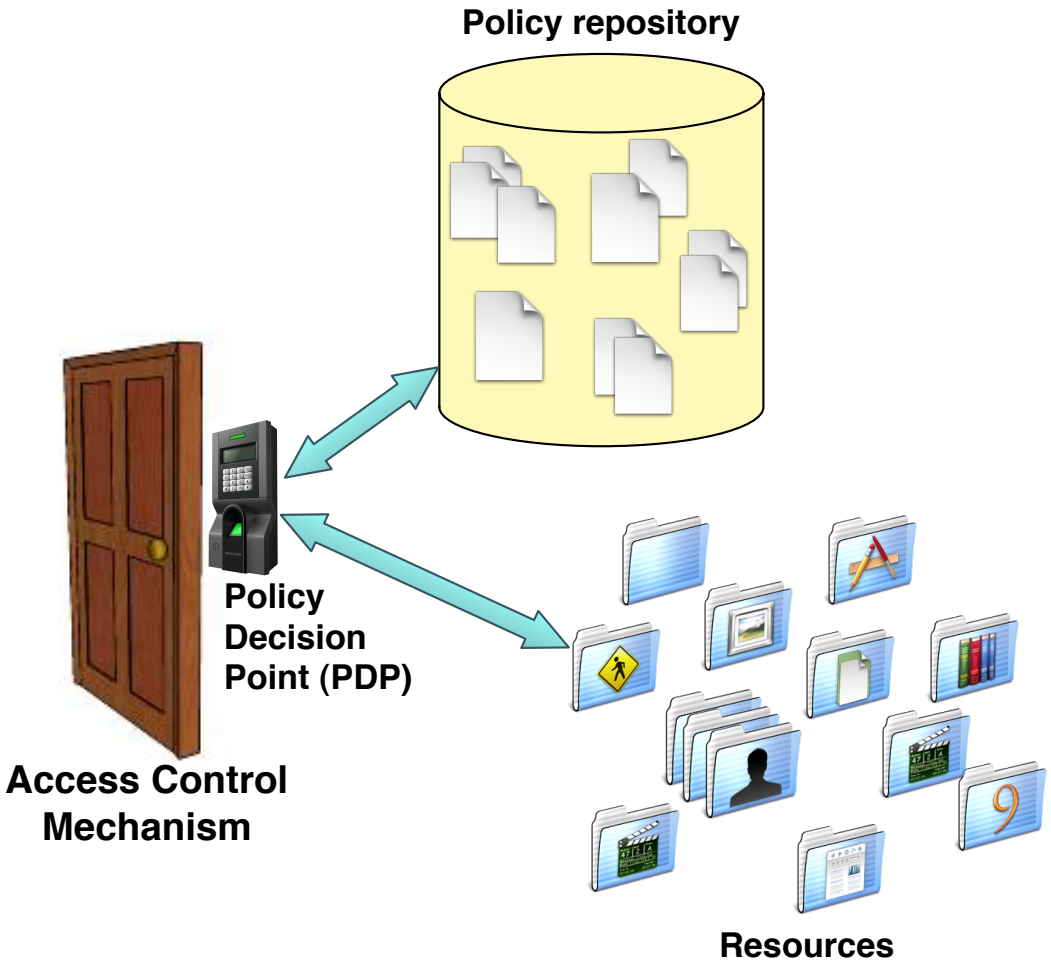
Access Control

unauthorized
user

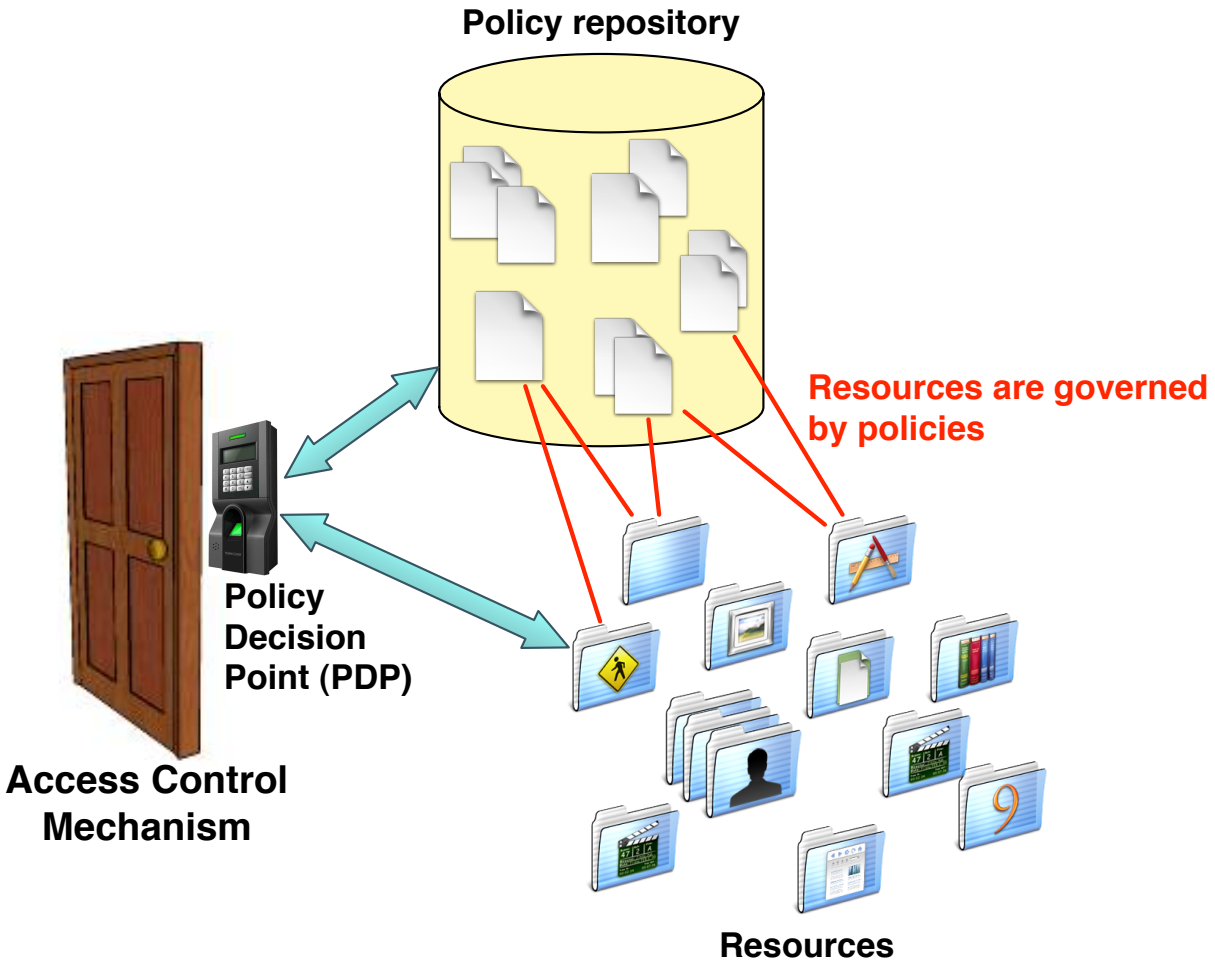


resource
(e.g. EHR)

Policy Based Access Control (PBAC)



Policy Based Access Control (PBAC)



A **policy** is a document that exactly specifies the credentials and requirements that a requestor must fulfill to access some resources

Policy Based Access Control (PBAC)

ascens

- PBAC is by now the de-facto standard model for enforcing access control policies in service-oriented architectures
- An authorization decision is based on the values of some attributes required to allow access to a resource according to policies created by the security administrator

Policy Based Access Control (PBAC)

ascens

- PBAC is by now the de-facto standard model for enforcing access control policies in service-oriented architectures
- An authorization decision is based on the values of some attributes required to allow access to a resource according to policies created by the security administrator
- Attributes are sets of properties used to describe the entities that must be considered for authorization; they might concern
  - the subject who is demanding access:

    identity, role, age, zip code, IP address, group memberships, citizenships, company, management level, certifications, etc.
  - the action that the subject wants to perform:

    read and/or write, patterns of argument data, etc.
  - the object (or resource) impacted by the action:

    identity, location, size, value, EHR, etc.

Policy Based Access Control (PBAC): advantages

ascens

- Overcomes scalability problems of previous AC models and enables systems integration
- Enables fine-grained access control: targets and requests can be expressed with greater freedom than is usually the case
  - The request format is not limited to the form subject-object-action
- Provides dynamic, context-aware access control
  - The authorization decision may depend also on context/environment attributes, not only on subject/resource ones

The XACML standard

eXtensible Access Control Markup Language (XACML) Version 3.0

Candidate OASIS Standard 01

26 September 2012

XACML

- is a widely used implementation of PBAC
- defines an XML-based language for writing *policies*
- defines an XML-based language for representing *access requests*
- defines how a PDP mades authorization decision
- is currently used in many large scale projects (e.g., epSOS, NHIN)

Policy Syntax

ascens

$$\Pi ::= \langle \mathit{Decision}\,;\,\mathsf{target}\!:\{\ \mathit{Targets}\ \}\rangle \quad \text{(Policies)}$$

$$\quad\quad \langle \mathit{Decision}\,;\,\mathsf{target}\!:\{\ \mathit{Targets}\ \}\rangle \quad \text{(atomic policy)}$$

$$\quad \mid\ \Pi\ \mathsf{p\text{-}o}\ \Pi \ \mid\ \Pi\ \mathsf{d\text{-}o}\ \Pi \quad \text{(policy combination)}$$

Policy Syntax

ascens

$$
\begin{array}{lll}
\Pi & ::= & \\
 & \quad \langle \mathit{Decision}\,;\, \mathsf{target}\colon \{\ \mathit{Targets}\ \} \rangle & \text{(atomic policy)} \\
 & \quad \mid \ \Pi\ \mathsf{p\text{-}o}\ \Pi \ \mid \ \Pi\ \mathsf{d\text{-}o}\ \Pi & \text{(policy combination)} \\
 & & \text{(Policies)} \\
 & & \\
\mathit{Decision} & ::= \ \mathsf{permit} \ \mid \ \mathsf{deny} & \text{(Decisions)}
\end{array}
$$

Policy Syntax

ascens

$$\Pi ::= \langle \mathit{Decision}\,;\,\text{target:}\{\ \mathit{Targets}\ \}\rangle \quad \text{(atomic policy)}$$
$$\mid\ \Pi\ \text{p-o}\ \Pi \quad \mid\ \Pi\ \text{d-o}\ \Pi \quad \text{(policy combination)}$$

(Policies)

$$\mathit{Decision} ::= \text{permit} \quad \mid \quad \text{deny}$$

(Decisions)

$$\mathit{Targets} ::= \mathit{MatchF}(\mathit{Designator}, \mathit{Expr}) \quad \text{(atomic target)}$$
$$\mid\ \mathit{Targets}\ \text{or}\ \mathit{Targets} \quad \mid\ \mathit{Targets}\ \text{and}\ \mathit{Targets} \quad \text{(target combination)}$$

(Targets)

Policy Syntax

ascens

$$
\begin{array}{rcll}
\Pi & ::= & & \text{(Policies)} \\
 & & \langle \mathit{Decision}\,;\,\text{target:}\{\ \mathit{Targets}\ \}\rangle & \text{(atomic policy)} \\
 & | & \Pi\ \text{p-o}\ \Pi \quad | \quad \Pi\ \text{d-o}\ \Pi & \text{(policy combination)} \\
 & & & \\
\mathit{Decision} & ::= & \text{permit} \quad | \quad \text{deny} & \text{(Decisions)} \\
 & & & \\
\mathit{Targets} & ::= & & \text{(Targets)} \\
 & & \mathit{MatchF}(\mathit{Designator},\mathit{Expr}) & \text{(atomic target)} \\
 & | & \mathit{Targets}\ \text{or}\ \mathit{Targets} \quad | \quad \mathit{Targets}\ \text{and}\ \mathit{Targets} & \text{(target combination)} \\
 & & & \\
\mathit{MatchF} & ::= & \text{equal} \quad | \quad \text{pattern-match} & \text{(Matching functions)} \\
 & | & \text{greater-than} \quad | \quad \ldots & \\
 & & & \\
\mathit{Designator} & ::= & \text{action} \quad | \quad \text{pattern} \quad | \quad \text{subject.}\mathit{attr} \quad | \quad \text{object.}\mathit{attr} & \text{(Designators)} \\
 & & & \\
\mathit{Expr} & ::= & \textbf{get} \quad | \quad \textbf{qry} \quad | \quad \textbf{put} \quad | \quad \textbf{fresh} \quad | \quad \textbf{new} & \text{(Expressions)} \\
 & | & \mathit{T} \quad | \quad \mathit{value} & \\
 & | & \text{subject.}\mathit{attr} \quad | \quad \text{object.}\mathit{attr} & \\
 & | & \text{not}\ \mathit{Expr} \quad | \quad \mathit{Expr}\ \text{or}\ \mathit{Expr} \quad | \quad \mathit{Expr}\ \text{and}\ \mathit{Expr} & \\
 & | & \mathit{Expr} + \mathit{Expr} \quad | \quad \mathit{Expr} \times \mathit{Expr} \quad | \quad \ldots & \\
 & | & \mathit{Expr} < \mathit{Expr} \quad | \quad \mathit{Expr} = \mathit{Expr} \quad | \quad \ldots &
\end{array}
$$

Request Syntax

ascens

SACPL requests (ranged over by ρ) are functions mapping *names* to *elements*, written as collections of pairs of the form (*name*, *element*)

Request Syntax

ascens

SACPL requests (ranged over by ρ) are functions mapping *names* to *elements*, written as collections of pairs of the form (*name*, *element*)

A typical example of request

$$\{(\text{subject}, \mathcal{I}), (\text{item}, t), (\text{action}, \textbf{get}), (\text{object}, \mathcal{J})\}$$

- Interface $\mathcal{I}$ is the subject of the request; it provides a set of attributes characterising the corresponding element
- $\mathcal{J}$ is the object of the request
- $\mathcal{I}$ requires the authorization to withdraw (**get**) the item t from the $\mathcal{J}$

Request Syntax

ascens

SACPL requests (ranged over by ρ) are functions mapping *names* to *elements*, written as collections of pairs of the form (*name*, *element*)

A typical example of request

$$\{(\text{subject}, \mathcal{I}), (\text{item}, t), (\text{action}, \textbf{get}), (\text{object}, \mathcal{J})\}$$

- Interface $\mathcal{I}$ is the subject of the request; it provides a set of attributes characterising the corresponding element
- $\mathcal{J}$ is the object of the request
- $\mathcal{I}$ requires the authorization to withdraw (**get**) the item t from the $\mathcal{J}$

Semantics: $\Pi \vdash \rho$

$\Pi \vdash \rho$ means that the authorization decision returned by Π in response to ρ is permit, i.e. access to the resource requested in ρ is granted by Π

Integration with SCEL

ascens

- Orthogonal aspects of components behaviour can be regulated by different kinds of policies, *enforced together* but *evaluated separately*
- The policy of a component $\mathcal{I}[\mathcal{K}, \Pi, P]$ can be better thought of as a tuple of policies, e.g. $(\Pi_i, \Pi_{ac}, \ldots)$
  - Π_i is an *interaction policy* regulating the interaction among processes inside a SCEL component
  - Π_{ac} is an *access control policy*, written in SACPL, regulating the access to the knowledge and resources of SCEL components

Integration with SCEL

ascens

- Orthogonal aspects of components behaviour can be regulated by different kinds of policies, *enforced together* but *evaluated separately*
- The policy of a component $\mathcal{I}[\mathcal{K}, \Pi, P]$ can be better thought of as a tuple of policies, e.g. $(\Pi_i, \Pi_{ac}, \ldots)$
  - Π_i is an *interaction policy* regulating the interaction among processes inside a SCEL component
  - Π_{ac} is an *access control policy*, written in SACPL, regulating the access to the knowledge and resources of SCEL components

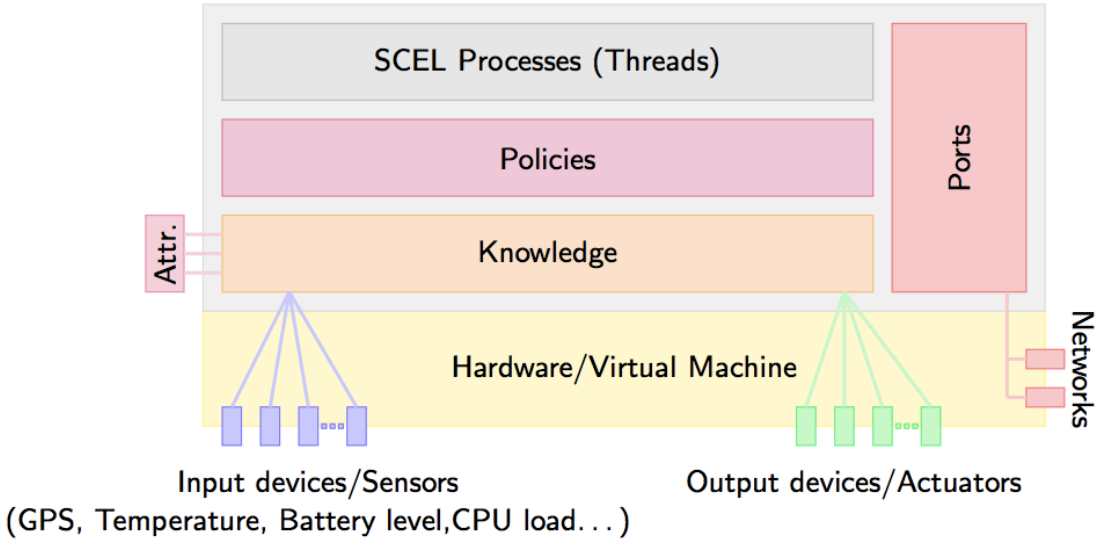
Authorization predicate $\Pi \vdash \lambda, \Pi'$

$$\frac{\Pi_{ac} \vdash \lambda 2\rho(\lambda)}{(\Pi_i, \Pi_{ac}, \ldots) \vdash \lambda, (\Pi_i, \Pi_{ac}, \ldots)}$$

The predicate converts authorization requests written in the SCEL's labels format into requests in the SACPL format through the function $\lambda 2\rho(\cdot)$, e.g.

$$\lambda 2\rho(\mathcal{I} : t \bar{\triangleleft} \mathcal{J}) \;=\; \{(\text{subject}, \mathcal{I}), (\text{item}, t), (\text{action}, \textbf{get}), (\text{object}, \mathcal{J})\}$$

A Run-time Environment for SCEL Programs



See next lecture (26 Sept) by Michele Loreti ...

For further details about SCEL, visit

http://rap.dsi.unifi.it/scel