ascens

Ensemble-oriented programming of self-adaptive systems with SCEL and jRESP

Michele Loreti

Dipartimento di Statistica, Informatica, Applicazioni
Università degli Studi di Firenze

AWASS 2013, Lucca, June 24-28, 2013

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

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

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
  - representing Knowledge, Behaviors and Aggregations according to specific Policies
  - programming interaction, adaptation and self- and context- awareness

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
  - representing Knowledge, Behaviors and Aggregations according to specific Policies
  - 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

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.

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

Key Notions

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

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

Key Notions

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

Key Notions

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

Key Notions

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.

The Service-Component Ensemble Language (SCEL) currently provides primitives and constructs for dealing with:

SCEL: programming abstractions

The Service-Component Ensemble Language (SCEL) currently provides primitives and constructs for dealing with:

1. Knowledge: to describe how data, information and (local and global) knowledge is managed

SCEL: programming abstractions

The Service-Component Ensemble Language (SCEL) currently provides primitives and constructs for dealing with:

1. Knowledge: to describe how data, information and (local and global) knowledge is managed
2. Behaviours: to describe how systems of components progress

SCEL: programming abstractions

The Service-Component Ensemble Language (SCEL) currently provides primitives and constructs for dealing with:

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*

SCEL: programming abstractions

The Service-Component Ensemble Language (SCEL) currently provides primitives and constructs for dealing with:

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

1. Knowledge

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

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,
- ...

1. Knowledge

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

Knowledge handling mechanisms

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

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.

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.

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).

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

2. Behaviors

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

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

2. Behaviors

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.

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

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

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

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

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.

Actions

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

ascens

Actions

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

Action Targets

$c ::= n \mid x \mid \mathsf{self} \mid \mathsf{P}$

ascens

Actions

$\textbf{get}(T)@c \;\mid\; \textbf{qry}(T)@c \;\mid\; \textbf{put}(t)@c \;\mid\; \textbf{fresh}(n) \;\mid\; \textbf{new}(\mathcal{I}, \mathcal{K}, \Pi, P)$

Action Targets

$c \;::=\; n \;\mid\; x \;\mid\; \mathsf{self} \;\mid\; \mathsf{P}$

Actions manage knowledge repositories by

- withdrawing information - $\textbf{get}(T)@c$,
- retrieving information - $\textbf{qry}(T)@c$
- adding information - $\textbf{put}(t)@c$

Actions operate on knowledge repository c and use T as a pattern to select knowledge items.

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.

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

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

Components

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

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

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

ascens

Components

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

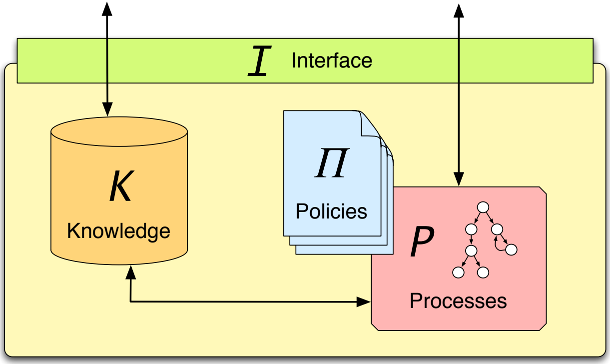
- An interface $\mathcal{I}$ containing information about the component itself. In particular, each component C has attributes:
  - *id*: the name of the component C
- A knowledge manager $\mathcal{K}$ providing control data (i.e. the local and (part of the) global knowledge) and application data; together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

Components

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

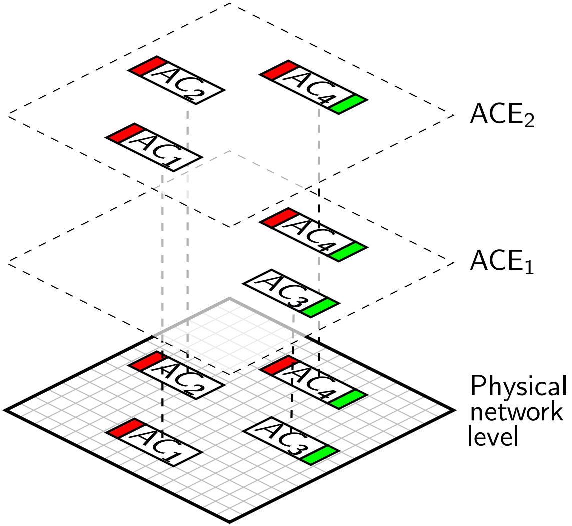
- An interface $\mathcal{I}$ containing information about the component itself. In particular, each component C has attributes:
  - *id*: the name of the component C
- A knowledge manager $\mathcal{K}$ providing control data (i.e. the local and (part of the) global knowledge) and application data; together 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

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)

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

Basic design principles. . .

Basic design principles...

1. no centralized control

Basic design principles...

1. no centralized control
2. heavy use of *recurrent patterns* to simply the development of specific
   - policies
   - knowledge
   - ...

ascens

Basic design principles...

1. no centralized control
2. heavy use of *recurrent patterns* to simply the development of specific
   - policies
   - knowledge
   - ...
3. use of *open technologies* to support the integration with other tools/frameworks
   - Argos
   - DEECo
   - ...

SCEL Syntax...

SYSTEMS:

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

COMPONENTS:

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

PROCESSES:

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

ACTIONS:

$$a \quad ::= \quad \textbf{get}(T)@c \quad | \quad \textbf{qry}(T)@c \quad | \quad \textbf{put}(t)@c \quad | \quad \textbf{exec}(P) \quad | \quad \textbf{new}(\mathcal{I}, \mathcal{K}, \Gamma$$

TARGETS:

$$c \quad ::= \quad n \quad | \quad x \quad | \quad \mathsf{self} \quad | \quad \mathsf{P}$$

SCEL Syntax...

SYSTEMS:

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

COMPONENTS:

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

PROCESSES:

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

ACTIONS:

$a \quad ::= \quad \mathbf{get}(T)@c \quad | \quad \mathbf{qry}(T)@c \quad | \quad \mathbf{put}(t)@c \quad | \quad \mathbf{exec}(P) \quad | \quad \mathbf{new}(\mathcal{I}, \mathcal{K}, \Gamma$

TARGETS:

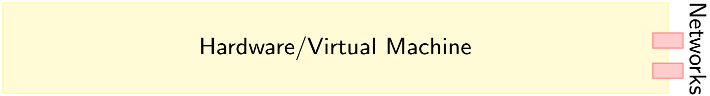
$c \quad ::= \quad n \quad | \quad x \quad | \quad \mathsf{self} \quad | \quad \mathsf{P}$

SCEL component

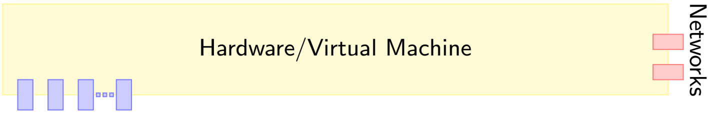
SCEL component

Hardware/Virtual Machine

SCEL component

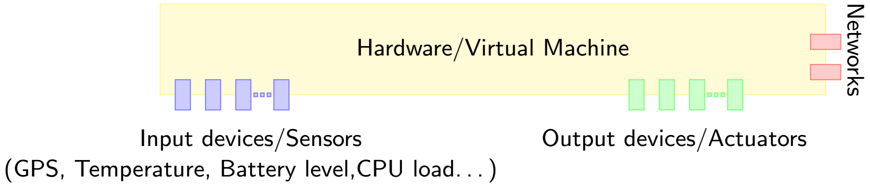


SCEL component

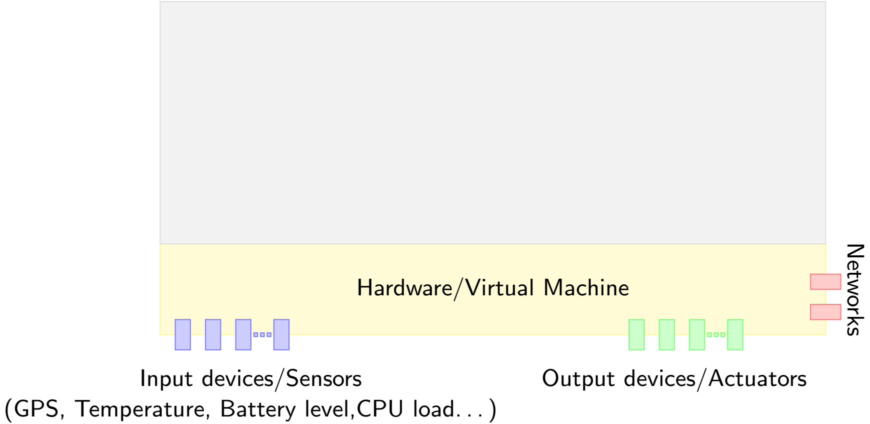


Input devices/Sensors
(GPS, Temperature, Battery level,CPU load. . .)

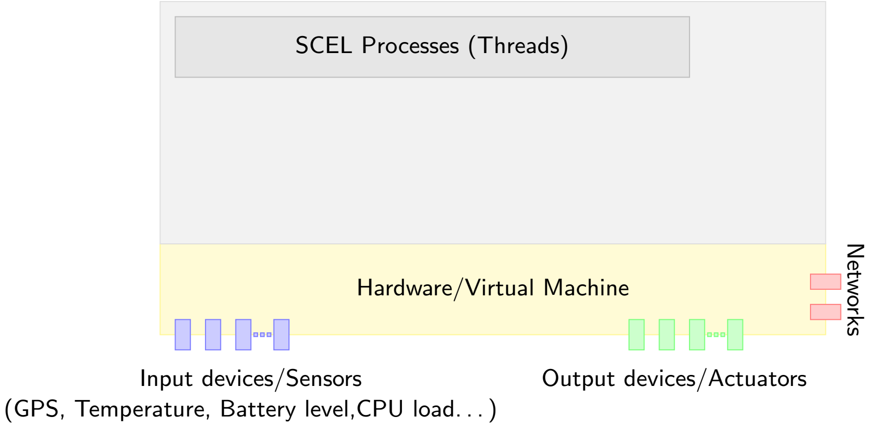
SCEL component



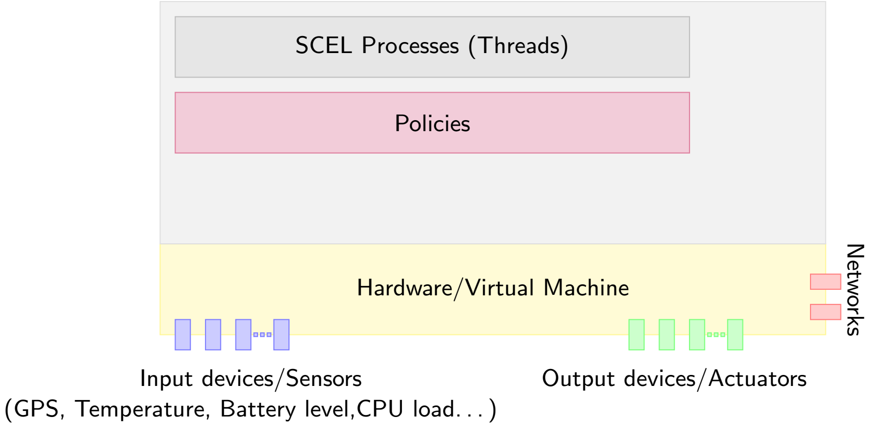
SCEL component



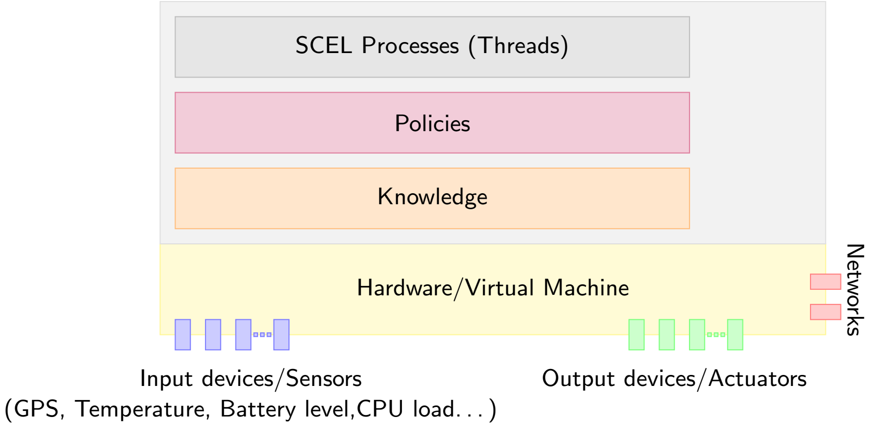
SCEL component

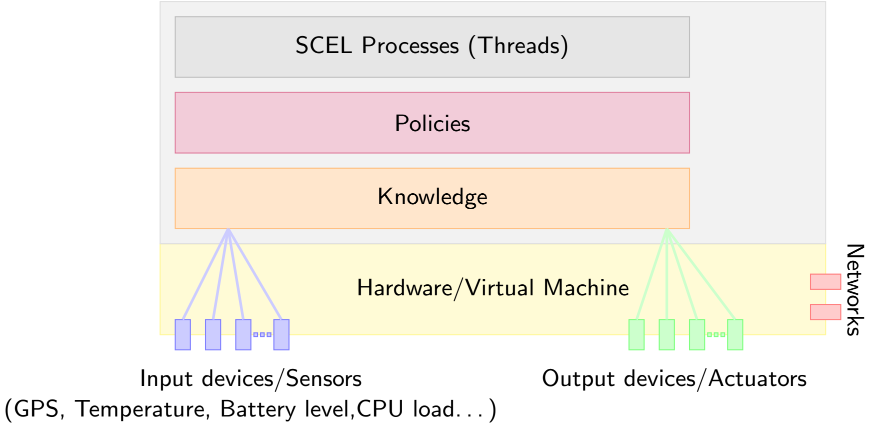


SCEL component

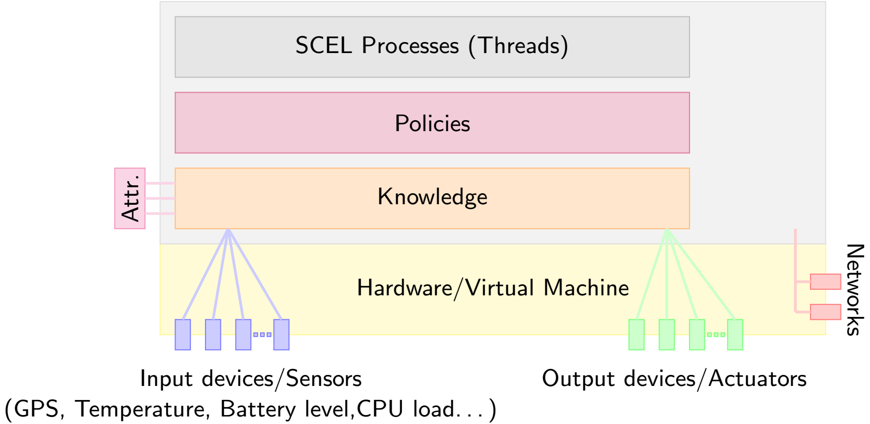


SCEL component

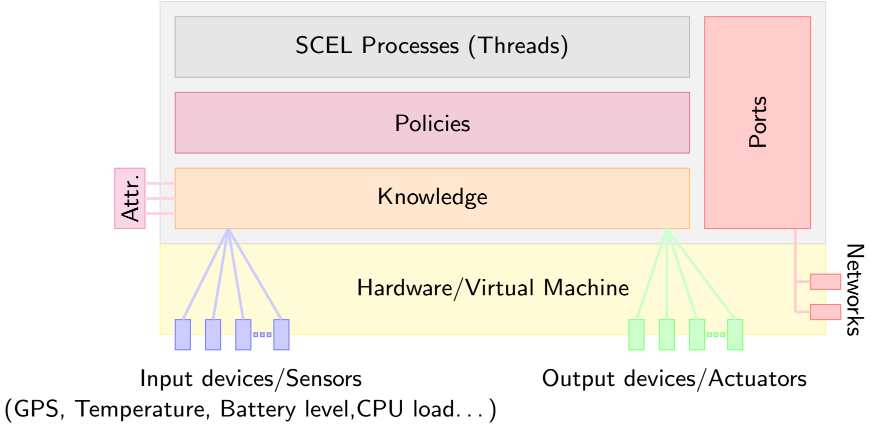



SCEL Processes (Threads)
Policies
Knowledge
Hardware/Virtual Machine
Networks
Input devices/Sensors
(GPS, Temperature, Battery level,CPU load...)
Output devices/Actuators

SCEL component



SCEL component



Knowledge

Knowledge

Components abstract from specific *Knowledge* implementations.

Knowledge

Components abstract from specific *Knowledge* implementations.

Indeed, *Knowledge* is a Java interface with the following methods:

- put(Tuple t): boolean
- get(Template t): Tuple
- query(Template t): Tuple

Knowledge

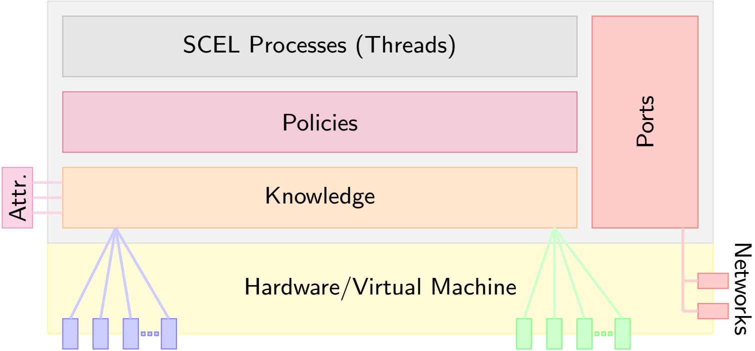
Components abstract from specific *Knowledge* implementations.

Indeed, *Knowledge* is a Java interface with the following methods:

- put(Tuple t): boolean
- get(Template t): Tuple
- query(Template t): Tuple

A single implementation of this interface is currently provided:

- class TupleSpace implements a *tuple space* (*á la klaim*)



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

Output devices/Actuators

Sensors. . .

Sensors. . .

Abstract class Sensor is used to identify a generic source of information:

- it can be associated to a logical/physical sensor
- values are exported as a *tuple*, each implementation has to define the structure of the tuple containing
  - (“GPS” , 45.8 , 37.2)
  - (“BATTERY” , %87)
- query actions are used to *retrieve* data from sensor

. . . and Actuators

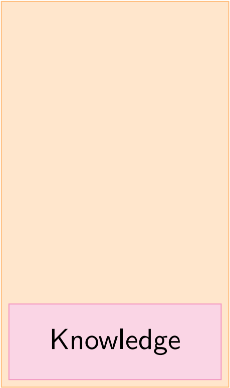
. . . and Actuators

Abstract class Actuator is used to identify an external device that can be controlled by SCEL processes:

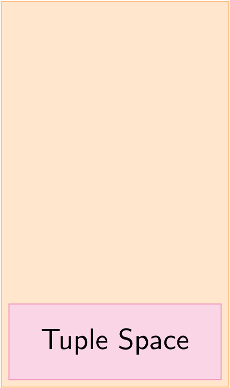
- it can be associated to a logical/physical actuator
- values are passed as a *tuple*, each implementation has to define the structure of the tuple containing
  - (“DIRECTION”, $\pi/3$)
- values are passed to a actuator via put actions

Component Knowledge in jRESP

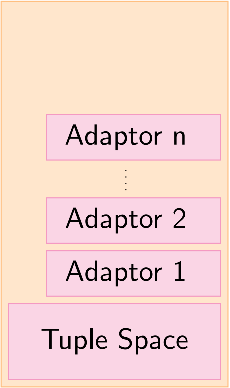
Component Knowledge in jRESP



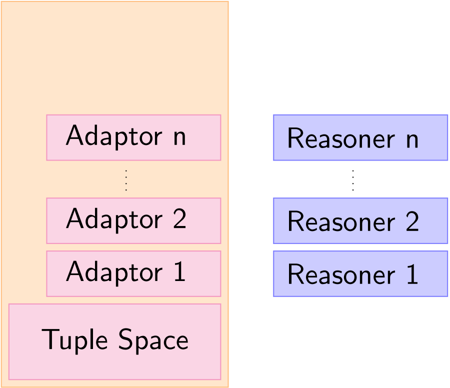
Component Knowledge in jRESP



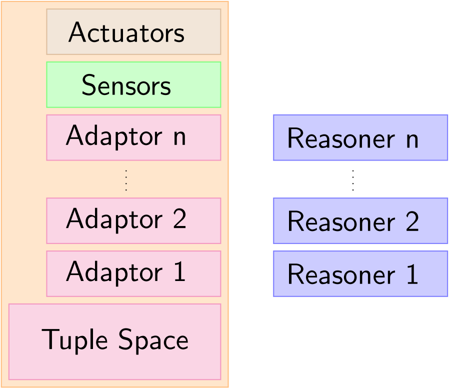
Component Knowledge in jRESP



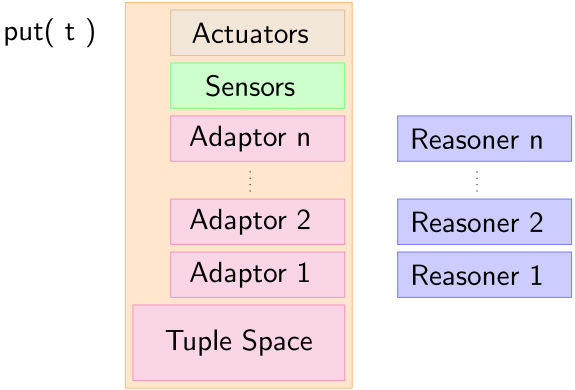
Component Knowledge in jRESP



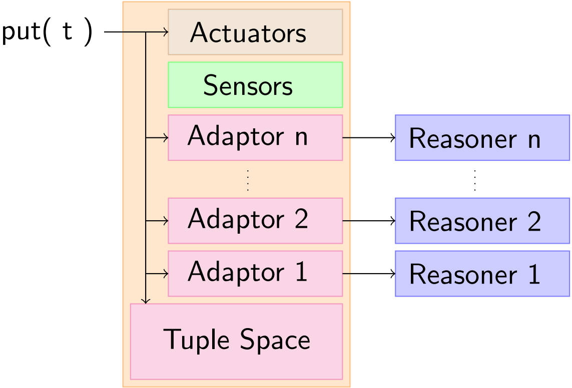
ascens


Actuators
Sensors
Adaptor n
Adaptor 2
Adaptor 1
Tuple Space
Reasoner n
Reasoner 2
Reasoner 1

Component Knowledge in jRESP

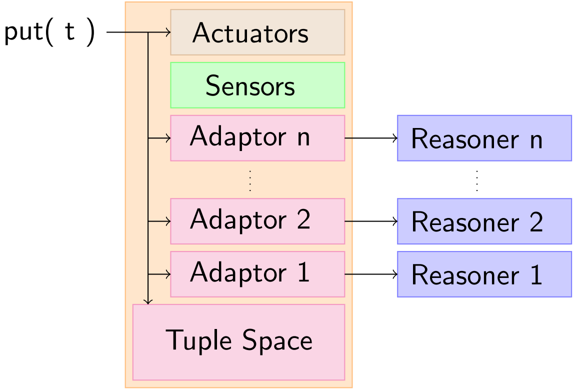


Component Knowledge in jRESP



When a put(t) is invoked, according to the structure of t, the *right* element is selected.

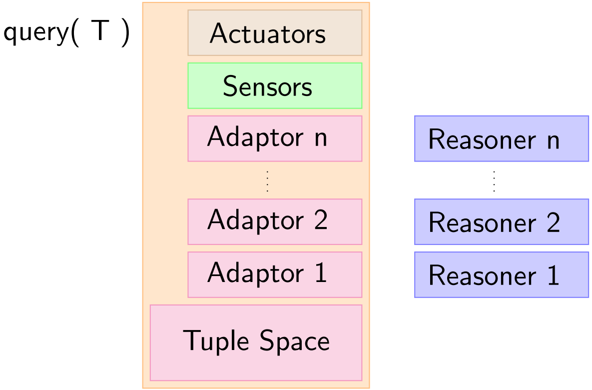
ascens

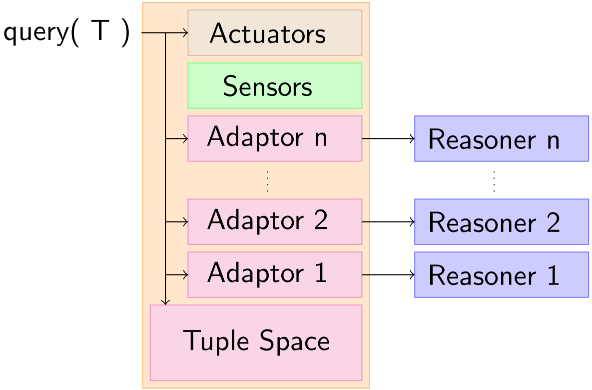


Adapters will use a *put* to add a knowledge item (e.g. a fact) represented by *t* in the corresponding reasoner.

Component Knowledge in jRESP

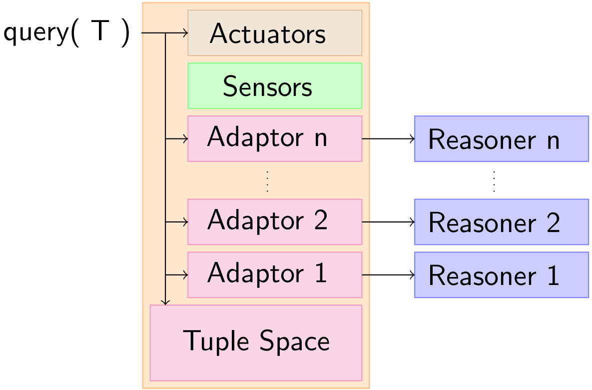
ascens





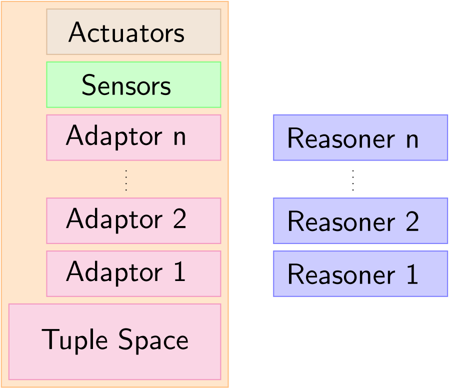
Similarly, a *get* will be *dispatched* according to template T.

Component Knowledge in jRESP



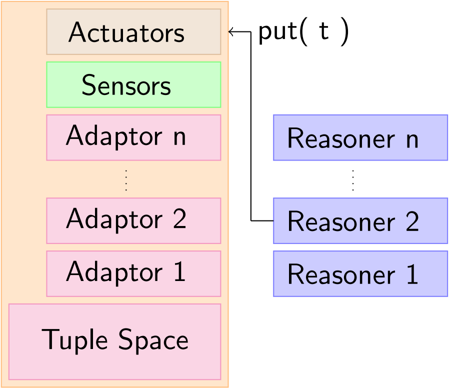
Adapters will use a *query* to ask to the corresponding reasoner about a *query* described via T.

Component Knowledge in jRESP



Reasoners can *autonomously* interact with the knowledge!

Component Knowledge in jRESP



Reasoners can *autonomously* interact with the knowledge!

Policies

Policies

Action execution on each component is regulated by a *policy*.

Policies

Action execution on each component is regulated by a *policy*.

This is an interface that provides methods like:

- put(Agent a, Tuple t , Target l)
- acceptGet(Locality l , Template t)

each node *forwards* received request to its policy.

Policies

Action execution on each component is regulated by a *policy*.

This is an interface that provides methods like:

- put(Agent a, Tuple t , Target l)
- acceptGet(Locality l , Template t)

each node *forwards* received request to its policy.

Policies

Action execution on each component is regulated by a *policy*.

This is an interface that provides methods like:

- put(Agent a, Tuple t , Target l)
- acceptGet(Locality l , Template t)

each node *forwards* received request to its policy.

Policies are organized in a *stack*:

- the policy at one level relies on the one at the level below to actually execute SCEL actions
- the policy at the lower level is the one that allows any operation

Ports

Ports

Each node is equipped with a set of *ports* that are able to handle:

- *point-to-point* interactions
- *group* interactions (*ensemble* oriented)

Ports

Each node is equipped with a set of *ports* that are able to handle:

- *point-to-point* interactions
- *group* interactions (*ensemble* oriented)

Currently the following ports have been developed:

Ports

Each node is equipped with a set of *ports* that are able to handle:

- *point-to-point* interactions
- *group* interactions (*ensemble* oriented)

Currently the following ports have been developed:

- InetPort, this kind of ports uses TCP to *point-to-point* interactions and UDP for the *group* ones

Ports

Each node is equipped with a set of *ports* that are able to handle:

- *point-to-point* interactions
- *group* interactions (*ensemble* oriented)

Currently the following ports have been developed:

- InetPort, this kind of ports uses TCP to *point-to-point* interactions and UDP for the *group* ones
- ServerPort, in this case a centralized server is used to collect and dispatch nodes' actions

Ports

Each node is equipped with a set of *ports* that are able to handle:

- *point-to-point* interactions
- *group* interactions (*ensemble* oriented)

Currently the following ports have been developed:

- InetPort, this kind of ports uses TCP to *point-to-point* interactions and UDP for the *group* ones
- ServerPort, in this case a centralized server is used to collect and dispatch nodes' actions
- VirtualPort, this is used to *simulate* nodes running on *virutal* devices

Ports

Each node is equipped with a set of *ports* that are able to handle:

- *point-to-point* interactions
- *group* interactions (*ensemble* oriented)

Currently the following ports have been developed:

- InetPort, this kind of ports uses TCP to *point-to-point* interactions and UDP for the *group* ones
- ServerPort, in this case a centralized server is used to collect and dispatch nodes' actions
- VirtualPort, this is used to *simulate* nodes running on *virutal* devices

To simplify interactions with other framework/tools, even developed in languages that are different from Java, JSon format is used.

Example: Group actions

Example: Group actions

get(T)@P

get(T)@P

1. A request message containing template T and attribute names occurring in P is sent via a *multicast* channel

Example: Group actions

get(T)@P

1. A request message containing template T and attribute names occurring in P is sent via a *multicast* channel
2. All the components that receives the message reply with:
   - a tuple t matching template T
   - a list of requested attributes

Example: Group actions

get(T)@P

1. A request message containing template T and attribute names occurring in P is sent via a *multicast* channel
2. All the components that receives the message reply with:
   - a tuple t matching template T
   - a list of requested attributes
3. If the node originating the requests receives attributes that *satisfies* predicate P, action is executed. Otherwise, the request is sent again.

P2P group-oriented protocol

P2P group-oriented protocol

We have integrated a new port based on *FreePastry* framework.

- point-to-point interactions relies on DHT primitives (the key is the node name);
- group-oriented interactions are implemented via multicast.

P2P group-oriented protocol

We have integrated a new port based on *FreePastry* framework.

- point-to-point interactions relies on DHT primitives (the key is the node name);
- group-oriented interactions are implemented via multicast.

Group-oriented via multicast:

P2P group-oriented protocol

We have integrated a new port based on *FreePastry* framework.

- point-to-point interactions relies on DHT primitives (the key is the node name);
- group-oriented interactions are implemented via multicast.

Group-oriented via multicast:

- We associate a *topic* to each predicate;

P2P group-oriented protocol

ascens

We have integrated a new port based on *FreePastry* framework.

- point-to-point interactions relies on DHT primitives (the key is the node name);
- group-oriented interactions are implemented via multicast.

Group-oriented via multicast:

- We associate a *topic* to each predicate;
- When a predicate is satisfied/unsatisfied a component register/deregister for a *topic*;

P2P group-oriented protocol

We have integrated a new port based on *FreePastry* framework.

- point-to-point interactions relies on DHT primitives (the key is the node name);
- group-oriented interactions are implemented via multicast.

Group-oriented via multicast:

- We associate a *topic* to each predicate;
- When a predicate is satisfied/unsatisfied a component register/deregister for a *topic*;
- Operations on a predicate are then realised via a multicast on the corresponding topic;

P2P group-oriented protocol

We have integrated a new port based on *FreePastry* framework.

- point-to-point interactions relies on DHT primitives (the key is the node name);
- group-oriented interactions are implemented via multicast.

Group-oriented via multicast:

- We associate a *topic* to each predicate;
- When a predicate is satisfied/unsatisfied a component register/deregister for a *topic*;
- Operations on a predicate are then realised via a multicast on the corresponding topic;
- A *special* topic is used to coordinate activities.

Robots in the swarm are distributed over a physical area and have to reach different zones according an assigned task:

- each robot has to fulfil one of two different tasks ($task_1$ or $tasks_2$).

Robots in the swarm are distributed over a physical area and have to reach different zones according an assigned task:

- each robot has to fulfil one of two different tasks ($task_1$ or $tasks_2$).

Robots are unaware about the position of the two target zones:

Swarm robotics scenario. . .

Robots in the swarm are distributed over a physical area and have to reach different zones according an assigned task:

- each robot has to fulfil one of two different tasks ($task_1$ or $tasks_2$).

Robots are unaware about the position of the two target zones:

- to discover the location of the target, robots follow a *random walk*;

Robots in the swarm are distributed over a physical area and have to reach different zones according an assigned task:

- each robot has to fulfil one of two different tasks ($task_1$ or $tasks_2$).

Robots are unaware about the position of the two target zones:

- to discover the location of the target, robots follow a *random walk*;
- when a robot reaches the area, it 'publishes' its location in the local knowledge repository. . .

Swarm robotics scenario...

Robots in the swarm are distributed over a physical area and have to reach different zones according an assigned task:

- each robot has to fulfil one of two different tasks ($task_1$ or $tasks_2$).

Robots are unaware about the position of the two target zones:

- to discover the location of the target, robots follow a *random walk*;
- when a robot reaches the area, it 'publishes' its location in the local knowledge repository...
  - robots with the same *task* can get aware about position of target area.

Swarm robotics scenario...

Robots in the swarm are distributed over a physical area and have to reach different zones according an assigned task:

- each robot has to fulfil one of two different tasks ($task_1$ or $tasks_2$).

Robots are unaware about the position of the two target zones:

- to discover the location of the target, robots follow a *random walk*;
- when a robot reaches the area, it 'publishes' its location in the local knowledge repository...
  - robots with the same *task* can get aware about position of target area.
- robots stop moving when the level of their batteries goes under a given threshold.

Ensembles in robotic scenario...

Ensembles in robotic scenario. . .

Q: What are the "ensembles" in the considered scenario?

Q: What are the "ensembles" in the considered scenario?

A: The set of robots that can fulfil the same task.

Ensembles in robotic scenario...

Q: What are the "ensembles" in the considered scenario?

A: The set of robots that can fulfil the same task.

NB: There is no central coordinator!

Ensembles in robotic scenario. . .

Q: What are the "ensembles" in the considered scenario?

A: The set of robots that can fulfil the same task.

NB: There is no central coordinator!

Q: How can "ensembles" be identified?

Ensembles in robotic scenario. . .

Q: What are the “ensembles” in the considered scenario?

A: The set of robots that can fulfil the same task.

NB: There is no central coordinator!

Q: How can “ensembles” be identified?

A: Each robot publishes in its interface the task that it can fulfil.

Ensembles in robotic scenario. . .

Q: What are the "ensembles" in the considered scenario?

A: The set of robots that can fulfil the same task.

NB: There is no central coordinator!

Q: How can "ensembles" be identified?

A: Each robot publishes in its interface the task that it can fulfil.

By relying on *group-oriented* queries robots in the same ensemble can get aware about the position of the zone and then can move directly towards the target.

Robotics scenario in SCEL. . .

Robotics scenario in SCEL. . .

Each robot is a SCEL component with two main behaviours:

- *managed element* (*ME*), that executes the next *control step* retrieved from local knowledge;
- *autonomic manager* (*AM*), that depending on info received from the environment puts in the knowledge the next *control step*.

Robotics scenario in SCEL...

Each robot is a SCEL component with two main behaviours:

- *managed element* (ME), that executes the next *control step* retrieved from local knowledge;
- *autonomic manager* (AM), that depending on info received from the environment puts in the knowledge the next *control step*.

Info from the environment includes:

Each robot is a SCEL component with two main behaviours:

- *managed element* (*ME*), that executes the next *control step* retrieved from local knowledge;
- *autonomic manager* (*AM*), that depending on info received from the environment puts in the knowledge the next *control step*.

Info from the environment includes:

- values retrieved from robot's sensors (GPS sensor, target sensor,...);

Robotics scenario in SCEL...

Each robot is a SCEL component with two main behaviours:

- *managed element* (*ME*), that executes the next *control step* retrieved from local knowledge;
- *autonomic manager* (*AM*), that depending on info received from the environment puts in the knowledge the next *control step*.

Info from the environment includes:

- values retrieved from robot's sensors (GPS sensor, target sensor,...);
- messages collected/received from other robots.

Robotics scenario in SCEL. . .

ascens

Each robot is a SCEL component with two main behaviours:

- *managed element* (*ME*), that executes the next *control step* retrieved from local knowledge;
- *autonomic manager* (*AM*), that depending on info received from the environment puts in the knowledge the next *control step*.

Info from the environment includes:

- values retrieved from robot's sensors (GPS sensor, target sensor,. . .);
- messages collected/received from other robots.

Self-adaptation is realised by exploiting SCEL higher-order features:

- autonomic manager *AM* implements the adaptation logic by replacing the control step code in knowledge repository.

A component is created when an object of class Node is instantiated:

```
Node<TupleSpace> n =
          new Node<TupleSpace>(
                    "Robot"+i,
                    new TupleSpace()
          );
n.addPort(vp);
```

A component is created when an object of class Node is instantiated:

```
Node<TupleSpace> n =
          new Node<TupleSpace>(
                    "Robot"+i,
                    new TupleSpace()
          );
n.addPort(vp);
```

Class Node is parametrized with respect a class that implements the interface characterising a *knowledge manager*

A component is created when an object of class Node is instantiated:

```
Node<TupleSpace> n =
          new Node<TupleSpace>(
                    "Robot"+i,
                    new TupleSpace()
          );
n.addPort(vp);
```

Class Node is parametrized with respect a class that implements the interface characterising a *knowledge manager*

- in the example above we use TupleSpace

Robotics scenario in SCEL. . .

Step 1: Creating components

A component is created when an object of class Node is instantiated:

```
Node<TupleSpace> n =
          new Node<TupleSpace>(
                    "Robot"+i,
                    new TupleSpace()
          );
n.addPort(vp);
```

Class Node is parametrized with respect a class that implements the interface characterising a *knowledge manager*

- in the example above we use TupleSpace

New *knowledge managers*, **for instance the one based on KnowLang**, can be easily integrated!

Knowledge...

The interface Knowledge identifies a generic knowledge repository and indicates the high-level primitives to manage pieces of relevant information coming from different sources.

The interface Knowledge identifies a generic knowledge repository and indicates the high-level primitives to manage pieces of relevant information coming from different sources.

This interface contains the methods for withdrawing/retrieving/adding piece of knowledge from/to a repository:

- put(Template t): Tuple
- get(Template t): Tuple
- query(Template t): Tuple

Knowledge...

The interface Knowledge identifies a generic knowledge repository and indicates the high-level primitives to manage pieces of relevant information coming from different sources.

This interface contains the methods for withdrawing/retrieving/adding piece of knowledge from/to a repository:

- put(Template t): Tuple
- get(Template t): Tuple
- query(Template t): Tuple

Sensors and Actuators:

- external data can be collected into a knowledge repository via *sensors*;
- *actuators* can be used to forward data to external components.

Robotics scenario in SCEL...

Step 2: Adding sensors and actuators

```
n.addActuator(scenario.getDirectionActuator(i));
n.addSensor(scenario.getLocationSensor(i));
n.addActuator(scenario.getStopActuator(i));
n.addSensor(scenario.getBatterySensor(i));
n.addSensor(scenario.getTargetSensor(i));
```

Robotics scenario in SCEL...

Step 2: Adding sensors and actuators

```
n.addActuator(scenario.getDirectionActuator(i));
n.addSensor(scenario.getLocationSensor(i));
n.addActuator(scenario.getStopActuator(i));
n.addSensor(scenario.getBatterySensor(i));
n.addSensor(scenario.getTargetSensor(i));
```

Remark:

In the code above, *scenario* is the Java classes modelling the *physical environment* where robots work.

Attribute collectors...

Attribute values are published on component interfaces via *attribute collectors*:

- when a request for an attribute is received, the corresponding collector is selected.
- node's knowledge is used to compute the actual attribute value.

Attribute collectors. . .

Attribute values are published on component interfaces via *attribute collectors*:

- when a request for an attribute is received, the corresponding collector is selected.
- node's knowledge is used to compute the actual attribute value.

Each attribute collector is associated with a *name* and a *template*:

- when the attribute is evaluated a tuple matching the template is retrieved (via a predicative query action);
- the retrieved tuple is used to compute the actual attribute value.

ascens

```
n.addAttributeCollector(
     new AttributeCollector(
          "task",
          new Template(
               new ActualTemplateField( "task"),
               new FormalTemplateField(Integer.class)
          )
     ) {
          @Override
          protected Attribute doEval(Tuple t) {
               return new Attribute(
                    "task",
                    t.getElementAt(Integer.class, 1));
          }
     }
);
```

Behaviours. . .

SCEL processes are implemented as threads via the abstract class Agent which provides the methods for:

- executing another agent,
- for generating fresh names,
- for instantiating a new component
- and for withdrawing/retrieving/adding information items from/to shared knowledge repositories.

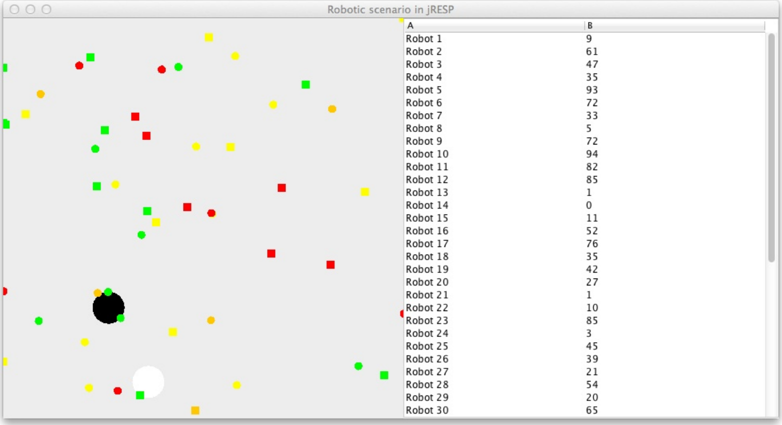
ascens

```
public class ManagedElement extends Agent {
   public ManagedElement() {
      super("ManagedElement");
   }
   @Override
   protected void doRun() throws Exception {
      while (true) {
         Tuple t = query(new Template(
                     new ActualTemplateField("controlStep")
                        ,
                     new FormalTemplateField(Agent.class)) ,
                  Self.SELF );
         Agent X = t.getElementAt(Agent.class, 1);
         call(X);
      }
   }
}
```

Autonomic manager (a fragment):

```
t = query( new Template(
        new ActualTemplateField("informed") ,
        new FormalTemplateField(Boolean.class)) ,
    Self.SELF );
boolean informed = t.getElementAt(Boolean.class, 1);
if (informed) {
    put( new Tuple( "controlStep" , new Informed() ) ,
        Self.SELF );
    get( new Template( new ActualTemplateField("seek") ) ,
        Self.SELF );
} else {
    put( new Tuple( "controlStep" , new RandomWalk() ) ,
        Self.SELF );
    get( new Template( new ActualTemplateField("seek") ) ,
        Self.SELF );
}
```

Running the scenario...



A	B
Robot 1	9
Robot 2	61
Robot 3	47
Robot 4	35
Robot 5	93
Robot 6	72
Robot 7	33
Robot 8	5
Robot 9	72
Robot 10	94
Robot 11	82
Robot 12	85
Robot 13	1
Robot 14	0
Robot 15	11
Robot 16	52
Robot 17	76
Robot 18	35
Robot 19	42
Robot 20	27
Robot 21	1
Robot 22	10
Robot 23	85
Robot 24	3
Robot 25	45
Robot 26	39
Robot 27	21
Robot 28	54
Robot 29	20
Robot 30	65

Concluding remarks

Concluding remarks

We have briefly presented (first version of) a framework that permits developing and executing SCEL oriented applications in Java

Concluding remarks

We have briefly presented (first version of) a framework that permits developing and executing SCEL oriented applications in Java

Considered framework should be now *populated* with...

Concluding remarks

We have briefly presented (first version of) a framework that permits developing and executing SCEL oriented applications in Java

Considered framework should be now *populated* with...

- specific implementations for *policies*, *knowledge* the proposed solutions;

Concluding remarks

ascens

We have briefly presented (first version of) a framework that permits developing and executing SCEL oriented applications in Java

Considered framework should be now *populated* with...

- specific implementations for *policies*, *knowledge* the proposed solutions;
- significative examples that can also help assessing/improving the performance of runtime framework.

Concluding remarks

We have briefly presented (first version of) a framework that permits developing and executing SCEL oriented applications in Java

Considered framework should be now *populated* with...

- specific implementations for *policies*, *knowledge* the proposed solutions;
- significative examples that can also help assessing/improving the performance of runtime framework.

We are now working on a *top-level* programming language that, enriching SCEL with standard programming primitives, permits simplifying development of SCEL programs.

Good work!