# Multi-Agent Oriented Programming


O. Boissier

Univ. Lyon, IMT Mines Saint-Etienne, LaHC UMR CNRS 5516, France

in tight collaboration with R.H. Bordini[2], J.F. Hübner[3], A. Ricci[4]

[2] Pontificia Universidade Catolica do Rio Grande do Sul (PUCRS), Porto Alegre, Brazil
[3] Federal University of Santa Catarina (UFSC), Florianópolis, Brazil
[4] University of Bologna (UNIBO), Bologna, Italy


Winter 2019

LABORATOIRE HUBERT CURIEN
UMR • CNRS • 5516 • SAINT-ETIENNE

MINES Saint-Étienne

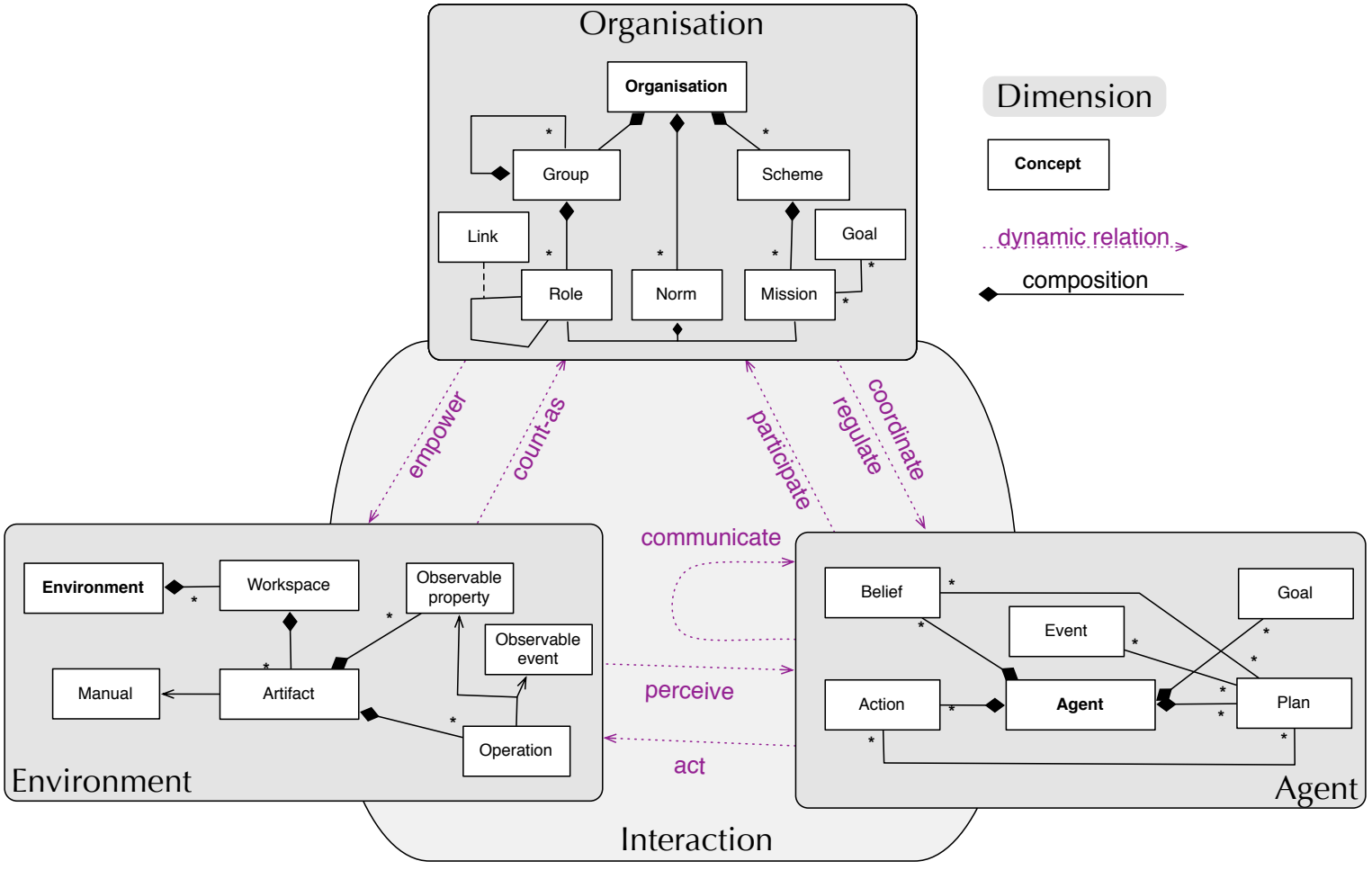
CONNECTED INTELLIGENCE

# Advanced Steps
# in
# Multi-Agent Oriented Programming

## Deeper Walk in JaCaMo Metamodel

# JaCaMo meta-model

Deeper walk in JaCaMo meta-model




Simplified view on JaCaMo meta-model [Boissier et al., 2011]

A seamless integration of three dimensions based on Jason [Bordini et al., 2007], Cartago [Ricci et al., 2009], Moise [Hübner et al., 2009] meta-models

# Outline

Programming Agents

Programming Environment

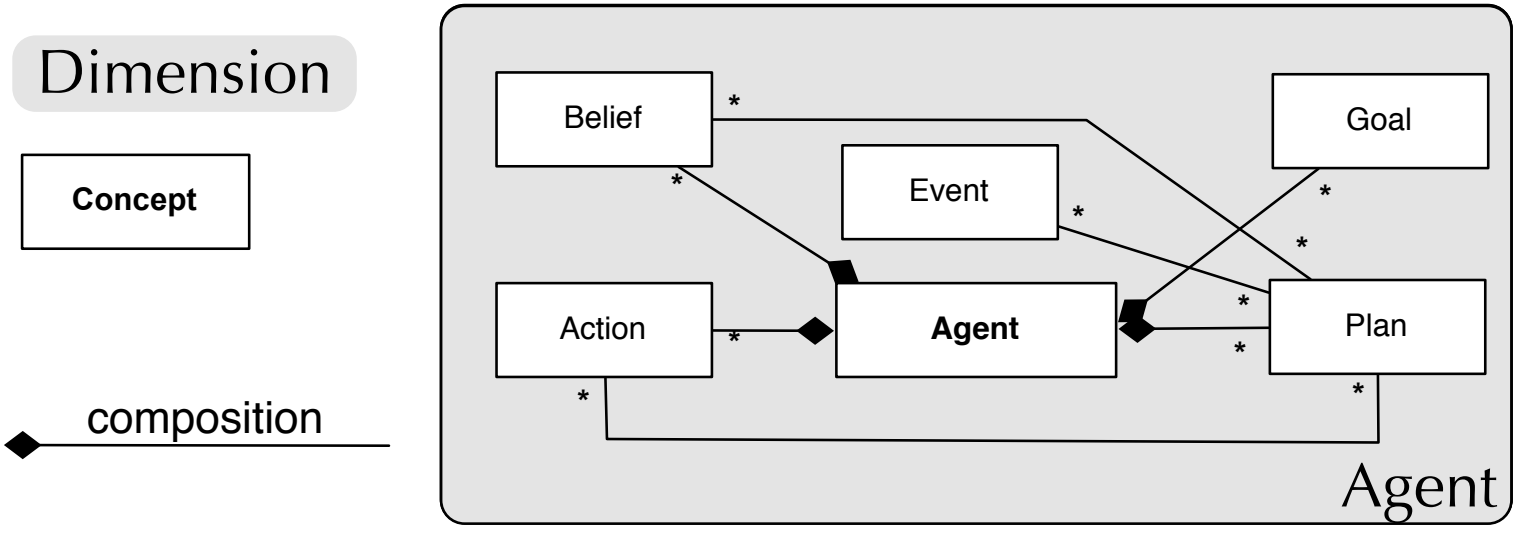
Programming Organization

Advanced practice of JaCaMo

Conclusion and Perspective

# Agent dimension

Programming Agents



# Agent program

Programming Agents

Main language constructs:

**Beliefs:** represent the information available to an agent (e.g. about the environment or other agents)

**Goals:** represent states of affairs the agent wants to bring about

**Plans:** are recipes for action, representing the agent's know-how

Actions can be internal, external, communicative or organisational ones

**Events:** happen as consequence to changes in the agent's beliefs or goals

**Intentions:** plans instantiated to achieve some goal

**Note:** identifiers starting in upper case denote variables

# Agent program

Programming Agents

Main language constructs:

- **Beliefs:** represent the information available to an agent (e.g. about the environment or other agents)
- **Goals:** represent states of affairs the agent wants to bring about
- **Plans:** are recipes for action, representing the agent's know-how
  Actions can be internal, external, communicative or organisational ones

Runtime structures:

- **Events:** happen as consequence to changes in the agent's beliefs or goals
- **Intentions:** plans instantiated to achieve some goal

**Note:** identifiers starting in upper case denote variables

# Beliefs representation



## Syntax

Beliefs are represented by annotated literals of first order logic

`functor(`$term_1$`, …, `$term_n$`)[`$annot_1$`, …, `$annot_m$`]`

## Example (belief base of agent Tom)

```
red(box1)[source(percept)].
friend(bob,alice)[source(bob)].
lier(alice)[source(self),source(bob)].
~lier(bob)[source(self)].
```

# Goals representation



## Syntax

Goals are represented as beliefs with a prefix:

- **!** to denote achievement goal (goal *to do*)
- **?** to denote test goal (goal *to know*)

## Example (Initial goal of agent Tom)

```
!write(book).
```

# Plans representation



## Syntax

An AgentSpeak plan has the following general structure:

triggering_event : context <- body.

where:

- triggering_event: events that the plan is meant to handle
- context: situations in which the plan can be used
- body: course of action to be used to handle the event if the context is believed to be true at the time a plan is being chosen to handle the event

# Plans representation – Triggering events



- Events happen as consequence to changes in the agent's beliefs or goals
- An agent reacts to events by executing plans

## Syntax

- belief addition: +b
- belief deletion: -b
- achievement-goal addition: +!g
- achievement-goal deletion: -!g
- test-goal addition: +?g
- test-goal deletion): -?g

# Plans representation – Context



Context is a boolean expression with the following operators:

## Syntax

- Boolean operators
  - **&** (and)
  - | (or)
  - **not** (not)
  - = (unification)
  - >, >= (relational)
  - <, <= (relational)
  - == (equals)
  - \== (different)
- Arithmetic operators
  - **+** (sum)
  - **-** (subtraction)
  - * (multiply)
  - **/** (divide)
  - **div** (divide – integer)
  - **mod** (remainder)
  - ** (power)

# Plans representation – Body



A plan body may contain:

- Belief operators
  - `+` (new belief)
  - `-` (dispose belief)
  - `-+` (update belief)
- Goal operators
  - `!` (new achievement sub-goal)
  - `?` (new test sub-goal)
  - `!!` (new achievement goal)
- External actions
- Internal actions
  - Unlike actions, internal actions do not change the environment
  - Encapsulate code to be executed as part of the agent reasoning cycle
  - Internal actions can be used for invoking legacy code
- Constraints

# Internal Actions



- Internal actions can be defined by the user in Java

  `libname.action_name(...)`

- Standard (pre-defined) internal actions have an empty library name
  - `.print(`$term_1$`,`$term_2$`,...)`
  - `.union(`$list_1$`, `$list_2$`, `$list_3$`)`
  - `.my_name(`*var*`)`
  - `.send(`*ag*`,`*perf*`,`*literal*`)`
  - `.intend(`*literal*`)`
  - `.drop_intention(`*literal*`)`

- Many others available for: printing, sorting, list/string operations, manipulating the beliefs/annotations/plan library, creating agents, waiting/generating events, etc.

# Plans representation



## Example

```
+rain : time_to_leave(T) & clock.now(H) & H >= T
   <- !g1;           // new sub-goal
      !!g2;          // new goal
      ?b(X);         // new test goal
      +b1(T-H);      // add mental note
      -b2(T-H);      // remove mental note
      -+b3(T*H);     // update mental note
      jia.get(X); // internal action
      X > 10;        // constraint to carry on
      close(door);// external action
      !g3[hard_deadline(3000)]. // goal with deadline
```

# Plan representation

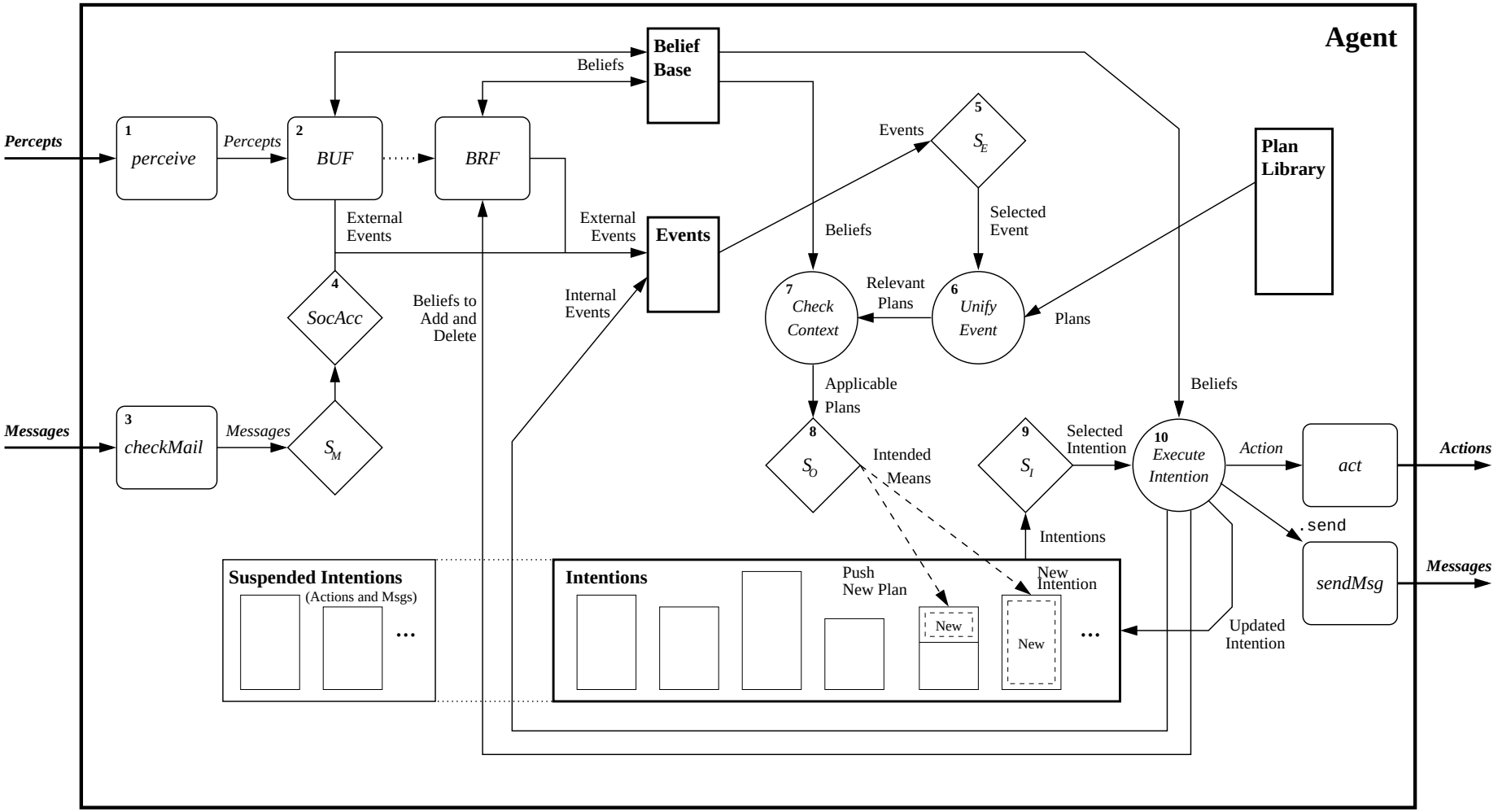


## Example

```
+green_patch(Rock)[source(percept)]
   :  not battery_charge(low)
   <- ?location(Rock,Coordinates);
      !at(Coordinates);
      !examine(Rock).

+!at(Coords)
   :  not at(Coords) & safe_path(Coords)
   <- move_towards(Coords);
      !at(Coords).
+!at(Coords)
   :  not at(Coords) & not safe_path(Coords)
   <- ...
+!at(Coords) : at(Coords).
```

# Agent dynamics

## Programming Agents



# Beliefs dynamics



## Internal reasoning

The plan operators + and - can be used to add and remove beliefs annotated with source(self) (mental notes)

```
+lier(alice); // adds lier(alice)[source(self)]
-lier(john); // removes lier(john)[source(self)]
```

## Perception (from the environment)

Beliefs are automatically updated accordingly to the perception of the agent (annotated with source(percept))

# Beliefs dynamics



## Communication (from other agents)

When an agent receives a *tell* (resp. *untell*) message, the content is a new belief (annotated with the sender of the message) (resp. belief corresponding to the content is deleted)

```
.send(tom,tell,lier(alice)); // sent by bob
// adds lier(alice)[source(bob)] in Tom's Belief Base
...
.send(tom,untell,lier(alice)); // sent by bob
// removes lier(alice)[source(bob)] from Tom's Belief Base
```

# Goals dynamics



## Internal reasoning

The plan operators **!**, **!!** and **?** are used to add a new goal (annotated with source(self))

```
...
// adds new achievement goal !write(book)[source(self)]
!write(book);

// adds new test goal ?publisher(P)[source(self)]
?publisher(P);
...
```

# Goals dynamics



## Communication of achievement goal

When an agent receives an *achieve* message, the content is a new achievement goal (annotated with the sender of the message)

```
.send(tom,achieve,write(book)); // sent by Bob
// adds new goal write(book)[source(bob)] for Tom
.send(tom,unachieve,write(book)); // sent by Bob
// removes goal write(book)[source(bob)] for Tom
```

## Communication of test goal

When an agent receives an *askOne* or *askAll* message, the content is a new test goal (annotated with the sender of the message)

```
.send(tom,askOne,published(P),Answer); // sent by Bob
// adds new goal ?publisher(P)[source(bob)] for Tom
// the response of Tom will unify with Answer
```

# Plans dynamics



The plans that form the plan library of the agent come from

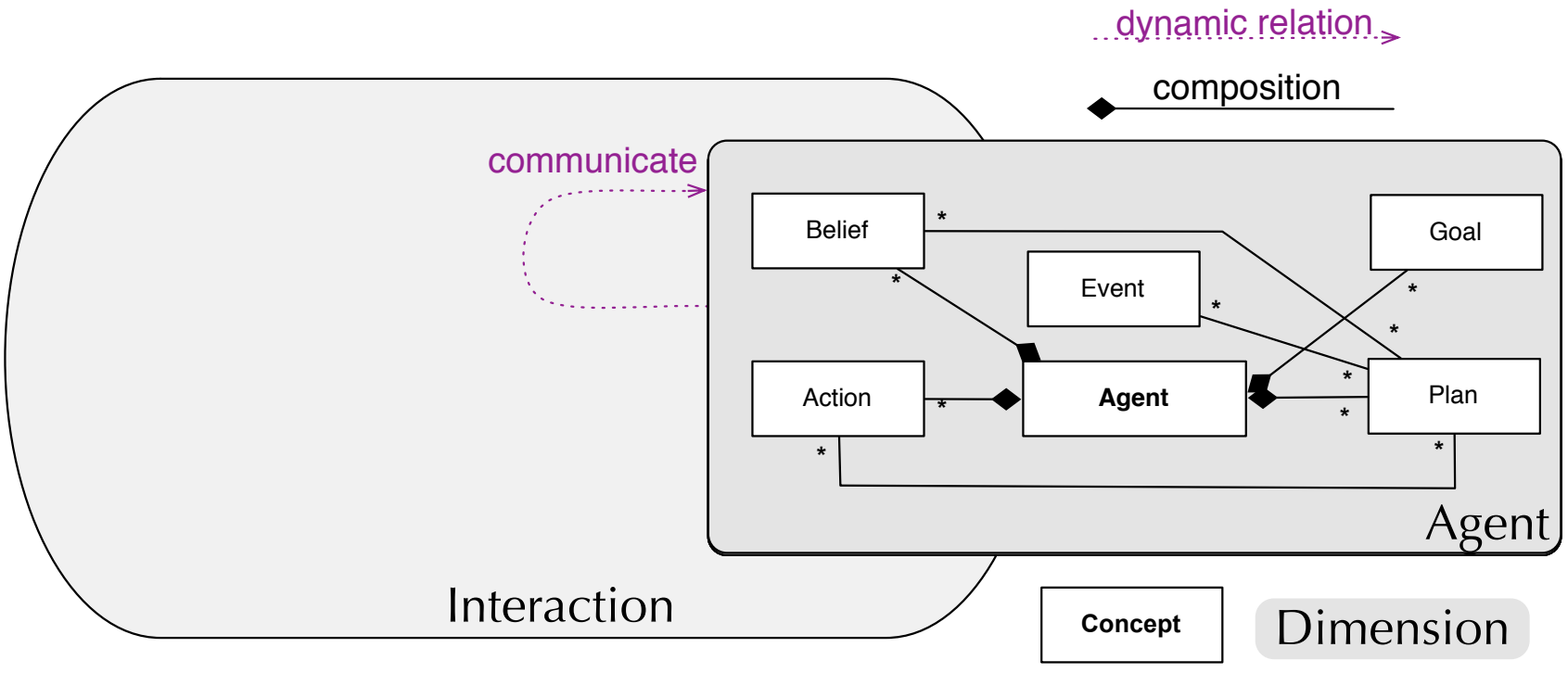
- plans added (resp. removed) dynamically by intentions in internal reasoning:
  - .add_plan (resp. .remove_plan)
- plans added (resp. removed) by communication:
  - *tellHow* (resp. *untellHow*)

## Example

```
.send(bob, askHow, +!goto(_,_)[source(_)], ListOfPlans);
...
.plan_label(Plan,hp); // get a plans based on a plan's label
.send(A,tellHow,Plan);
.send(bob,tellHow,"+!start : true <- .println(ẗhello)̈.").
```

# Integrating A & A dimensions

Deeper walk in JaCaMo meta-model



# Outline

Programming Agents

Programming Environment

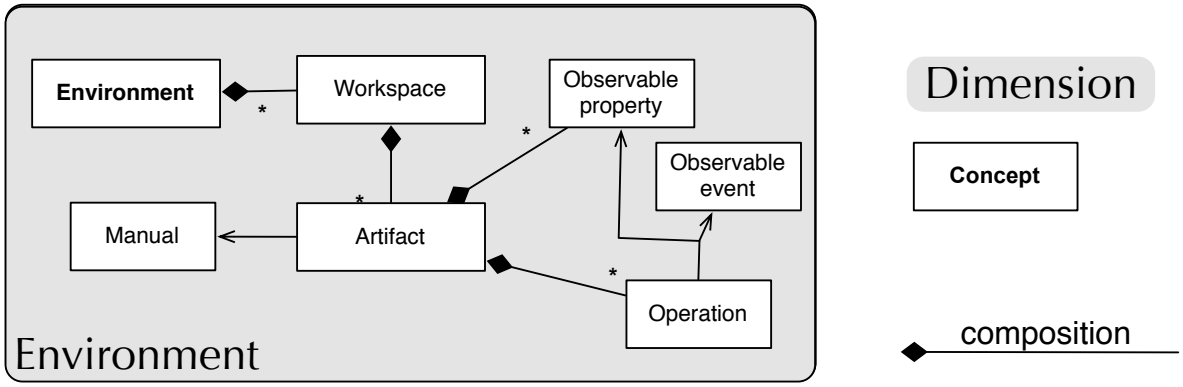
Programming Organization

Advanced practice of JaCaMo

Conclusion and Perspective

# Environment dimension

Programming Environment

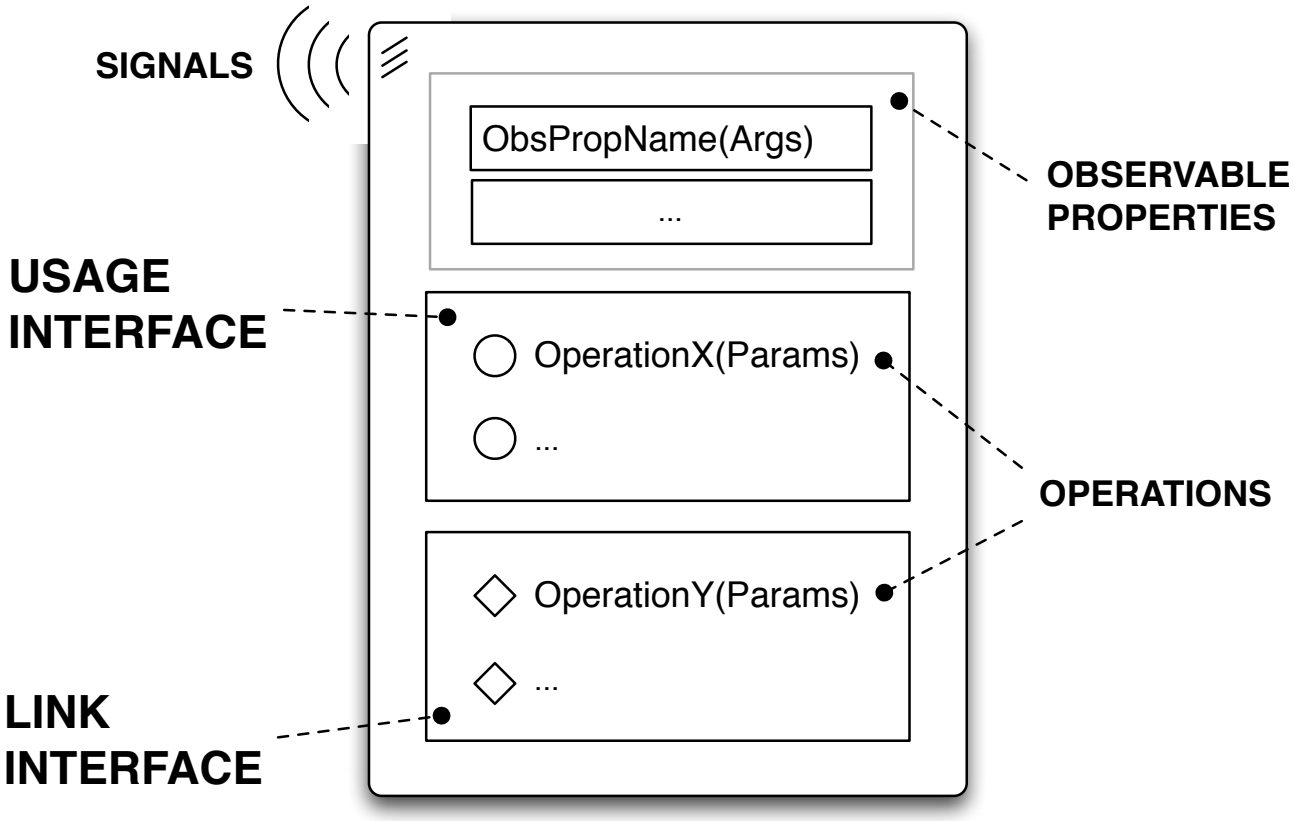



Based on A&A [Omicini et al., 2008]

- Artifacts
  - non-autonomous, function-oriented, stateful entities
    - controllable and observable
  - modelling the tools and resources used by agents
    - designed by MAS programmers
- Workspaces
  - grouping agents & artifacts
  - defining the topology of the computational environment

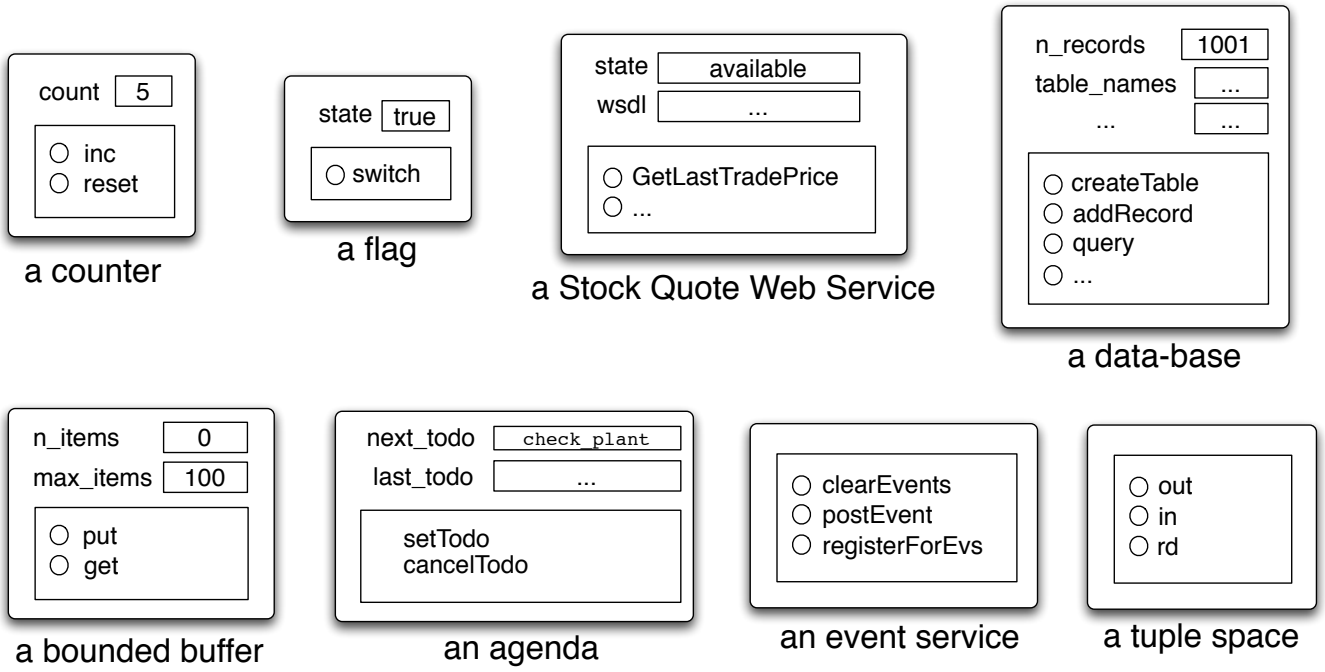
# Artifact Abstract Representation

Programming Environment



# A World of Artifacts

Programming Environment



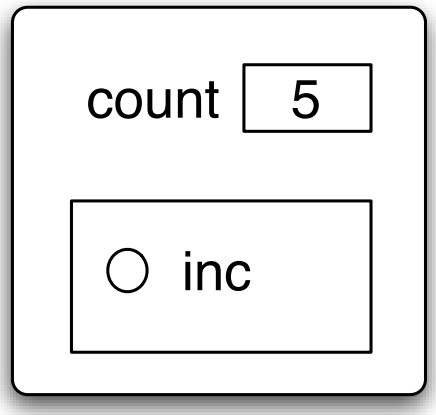

- Individual or personal artifacts: functionalities for a single agent use (e.g. agenda, library)
- Social artifacts: functionalities for structuring and managing the interaction (e.g. a blackboard, a game-board)
- Boundary artifacts: access external resources/services (e.g. a printer, a Web Service) or to represent devices enabling I/O with users (e.g GUI, console)

# Programming Artifacts – Some API spots

Programming Environment

- `Artifact` base class
- `@OPERATION` annotation to mark artifact's operations
- set of primitives to define/update/... observable properties
- primitive to generate signals: `signal`

```
class Counter extends Artifact {

  void init(){
    defineObsProp("count",0);
  }

  @OPERATION void inc(){
    ObsProperty p = getObsProperty("count");
    p.updateValue(p.intValue() + 1);
    signal("tick");
  }
}
```

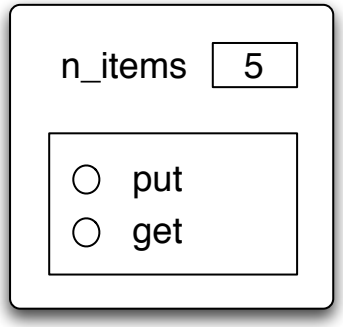


# Programming Artifacts – Basic operation features

## Programming Environment

- output parameters to represent action feedbacks
- long-term operations, with a high-level support for synchronization (await, @GUARD)

```
public class BoundedBuffer extends Artifact {
  private LinkedList<Item> items;
  private int nmax;

  void init(int nmax){
    items = new LinkedList<Item>();
    defineObsProperty("n_items",0);
    this.nmax = nmax;
  }

  @OPERATION void put(Item obj){
    await("bufferNotFull");
    items.add(obj);
    getObsProperty("n_items").updateValue(items.size());
  }

  @OPERATION void get(OpFeedbackParam<Item> res) {
    await("itemAvailable");
    Item item = items.removeFirst();
    res.set(item);
    getObsProperty("n_items").updateValue(items.size());
  }

  @GUARD boolean itemAvailable(){ return items.size() > 0; }

  @GUARD boolean bufferNotFull(Item obj){ return items.size() < nmax; }
}
```



# Programming Artifacts – Internal operations

Programming Environment

- execution of operations triggered by other operations
- implementing controllable *processes*

CLOCK

```
public class Clock extends Artifact {

  boolean working;
  final static long TICK_TIME = 100;

  void init(){ working = false; }

  @OPERATION void start(){
    if (!working){
      working = true;
      execInternalOp("work");
    } else {
      failed("already_working");
    }
  }

  @OPERATION void stop(){ working = false; }

  @INTERNAL_OPERATION void work(){
    while (working){
      signal("tick");
      await_time(TICK_TIME);
    }
  }
}
```

CLOCK USER AGENT

```
!test_clock.

+!test_clock
  <- makeArtifac("myClock","Clock",[],Id);
     focus(Id);
     +n_ticks(0);
     start;
     println("clock started.").

@plan1
+tick: n_ticks(10)
  <- stop;
     println("clock stopped.").

@plan2 [atomic]
+tick: n_ticks(N)
  <- -+n_ticks(N+1);
     println("tick perceived!").
```

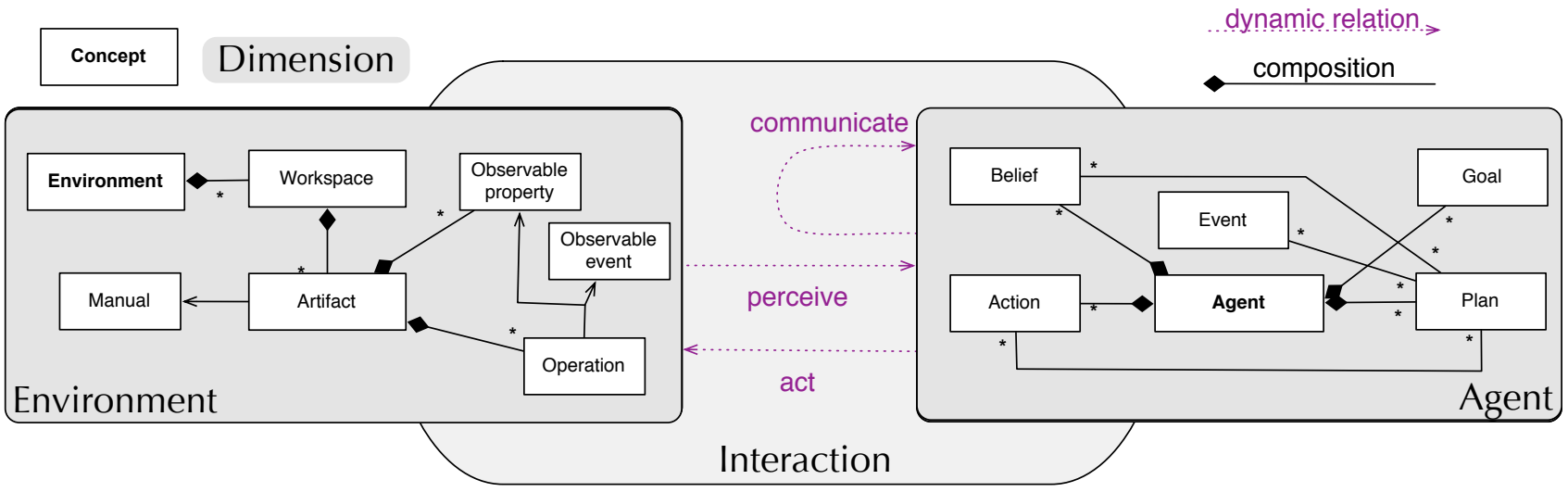
# Artifact dynamics

Programming Environment

- Process-based operation execution semantics
  - atomicity and transactionality: operations are executed transactionally with respect to the observable state of the artifact
  - concurrency: operations execution can overlap (use of the await primitive to break the operation in multiple transactional steps)
  - key feature for implementing coordination functionalities
- Observable property creation/change, when:
  - the operation completes, successfully
  - a signal is generated
  - the operation is suspended (by means of an await)
  - If an operation fails, changes to the observable state of the artifact are rolled back.
- Events creation

# Integrating A & E dimensions

Deeper walk in JaCaMo meta-model




Mapping of:

- Artifacts' operations onto agent actions
- Artifacts' observable properties onto agent beliefs
- Artifacts' signals onto belief-update events related to observable events
- *Jason* data-model is extended to manage also (Java) objects

Dynamic actions repertoire:

- given by the dynamic set of operations provided by the overall set of artifacts available in the workspace
- can be changed by creating/disposing artifacts

# Predefined actions repertoire

Deeper walk in JaCaMo meta-model

Accessible to agents through a predefined set of artifacts contained by default in each workspace:

- Functionalities to manage the workspace (including security):
  - operations: `makeArtifact`, `lookupArtifact`, `focus`,...
  - accessible through `workspace`, type: `cartago.WorkspaceArtifact`
- Core functionalities related to a node:
  - operations: `createWorkspace`, `joinWorkspace`, ...
  - accessible through `node`, type: `cartago.NodeArtifact`
- Others:
  - operations: `println`,... accessible through `console`, type `cartago.tools.Console`
  - operations: `out`, `in`, `rd`, ... accessible through `blackboard`, type `cartago.tools.TupleSpace`

$\rightsquigarrow$ pre-defined beliefs repertoire also

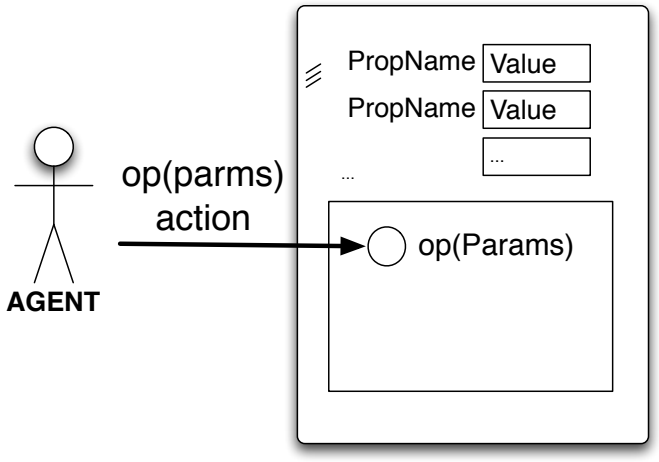
# Interaction model – Action execution

Deeper walk in JaCaMo meta-model

- Action success/failure semantics is defined by operation semantics
- Executing an action suspends the intention until completion or failure of the corresponding operation
  - Action completion events generated by the environment and automatically processed by the agent/environment platform bridge
  - No need of explicit observation and reasoning by agents to know if an action succeeded
- *The agent execution cycle is not blocked*!
  - The agent can continue to process percepts and possibly execute actions of other intentions

# Interaction Model: Use

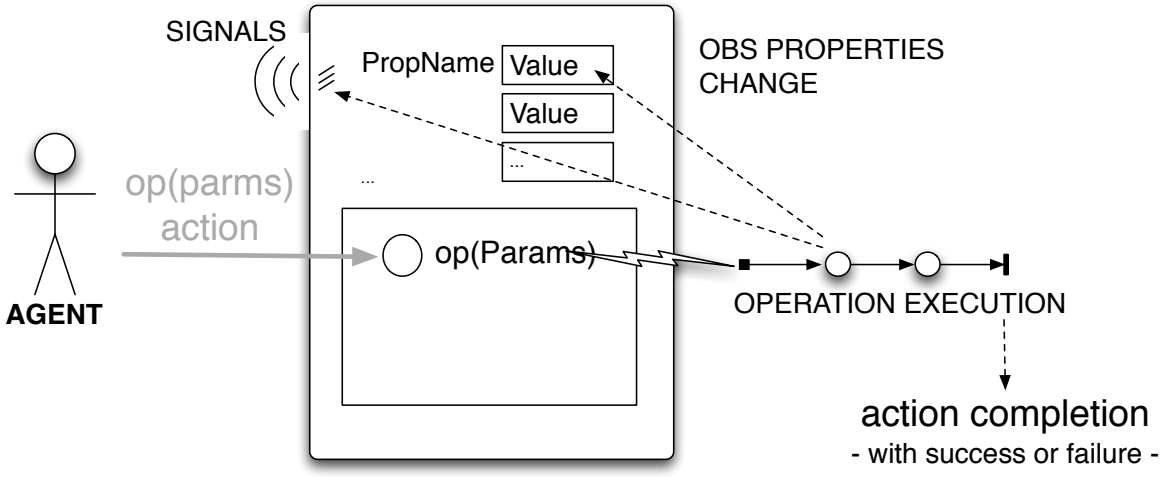
Deeper walk in JaCaMo meta-model




- Performing an action corresponds to triggering the execution of an operation
  - acting on artifact's usage interface

# Interaction Model: Operation execution

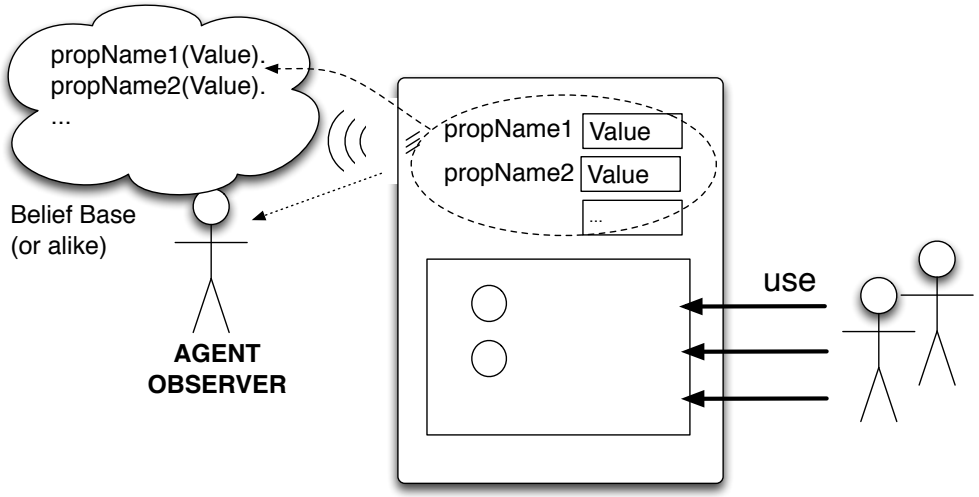
Deeper walk in JaCaMo meta-model




- A process structured in one or multiple transactional steps
- Asynchronous with respect to agent
  - *...which can proceed possibly reacting to percepts and executing actions of other plans/activities*
- Operation completion causes action completion
  - Action completion events with success or failure, possibly with action feedbacks

# Interaction Model: Observation

Deeper walk in JaCaMo meta-model

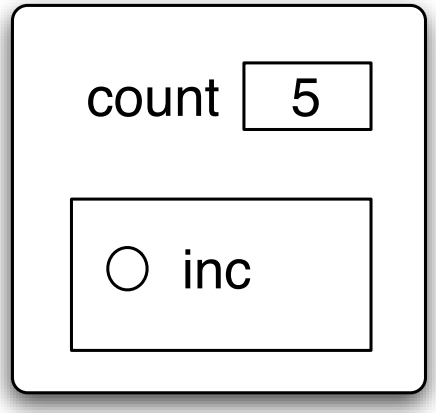



- Agents can dynamically select which artifacts to observe
  - `focus`/`stopFocus` actions
- By focussing an artifact
  - observable properties are mapped into agent dynamic knowledge about the state of the world, as percepts
    - e.g. belief base
  - signals are mapped as percepts related to observable events

# Example 1

Deeper walk in JaCaMo meta-model

```
class Counter extends Artifact {

  void init(){
    defineObsProp("count",0);
  }

  @OPERATION void inc(){
    ObsProperty p = getObsProperty("count");
    p.updateValue(p.intValue() + 1);
    signal("tick");
  }
}
```




User and Observer Agents Working with the shared counter

# Example 1

Deeper walk in JaCaMo meta-model

USER(S)

```
!create_and_use.

+!create_and_use : true
  <- !setupTool(Id);
     // use
     inc;
     // second use specifying the Id
     inc [artifact_id(Id)].

// create the tool
+!setupTool(C): true
  <- makeArtifact("c0","Counter",C).
```

OBSERVER(S)

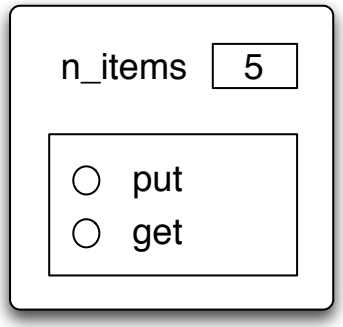
```
!observe.

+!observe : true
  <- ?myTool(C);  // discover the tool
     focus(C).

+count(V)
  <- println("observed new value: ",V).

+tick [artifact_name(Id,"c0")]
  <- println("perceived a tick").

+?myTool(CounterId): true
  <- lookupArtifact("c0",CounterId).

-?myTool(CounterId): true
  <- .wait(10);
     ?myTool(CounterId).
```

User and Observer Agents Working with the shared counter

# Example 2

Deeper walk in JaCaMo meta-model

```
public class BoundedBuffer extends Artifact {
  private LinkedList<Item> items;
  private int nmax;

  void init(int nmax){
    items = new LinkedList<Item>();
    defineObsProperty("n_items",0);
    this.nmax = nmax;
  }

  @OPERATION void put(Item obj){
    await("bufferNotFull");
    items.add(obj);
    getObsProperty("n_items").updateValue(items.size());
  }

  @OPERATION void get(OpFeedbackParam<Item> res) {
    await("itemAvailable");
    Item item = items.removeFirst();
    res.set(item);
    getObsProperty("n_items").updateValue(items.size());
  }

  @GUARD boolean itemAvailable(){ return items.size() > 0; }

  @GUARD boolean bufferNotFull(Item obj){ return items.size() < nmax; }
}
```




Producers and Consumers with Bounded buffer

# Example 2

Deeper walk in JaCaMo meta-model

PRODUCERS

```
item_to_produce(0).
!produce.

+!produce: true
  <- !setupTools(Buffer);
     !produceItems.

+!produceItems : true
  <- ?nextItemToProduce(Item);
     put(Item);
     !!produceItems.

+?nextItemToProduce(N) : true
  <- -item_to_produce(N);
     +item_to_produce(N+1).

+!setupTools(Buffer) : true
  <- makeArtifact("myBuffer","BoundedBuffer",
                  [10],Buffer).

-!setupTools(Buffer) : true
  <- lookupArtifact("myBuffer",Buffer).
```

CONSUMERS

```
!consume.

+!consume: true
  <- ?bufferReady;
     !consumeItems.

+!consumeItems: true
  <- get(Item);
     !consumeItem(Item);
     !!consumeItems.

+!consumeItem(Item) : true
  <- .my_name(Me);
     println(Me,": ",Item).

+?bufferReady : true
  <- lookupArtifact("myBuffer",_).
-?bufferReady : true
  <-.wait(50);
     ?bufferReady.
```

Producers and Consumers with Bounded buffer

# Outline

Programming Agents

Programming Environment

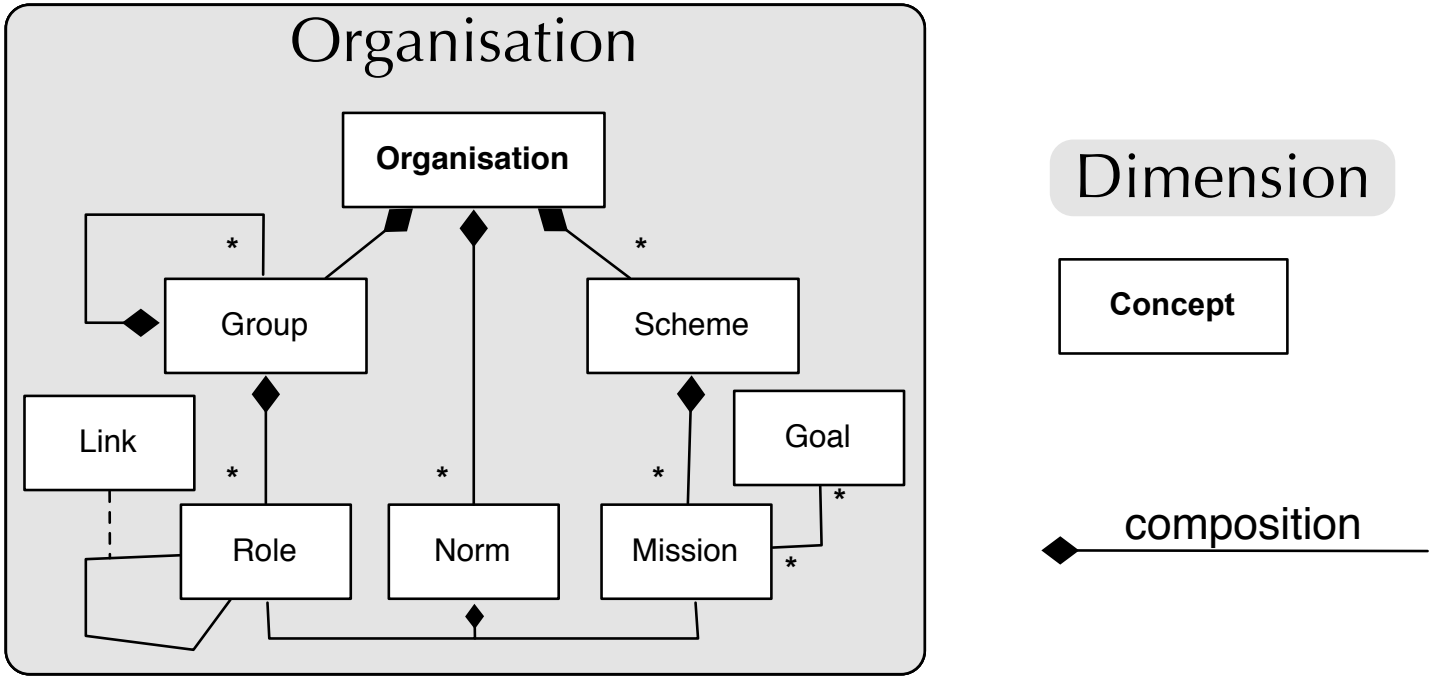
Programming Organization

Advanced practice of JaCaMo

Conclusion and Perspective

# Organization dimension

Programming Organization




organisational-specification section with three sub-sections: structural, functional, normative

# Structural specification

Programming Organization

- *Individual* level:
  - *Role*: label used to assign constraints on agents playing it
- *Collective* level:
  - *Group*: set of links, roles, compatibility relations used to define a shared context for agents playing roles in it
  - *Link*: relation between roles that directly constrains the agents in their interaction with the other agents playing the corresponding roles

## Example

```
<organisational-specification
  <structural-specification>
     <role-definitions>  ... </role-definitions>
     <group-specification id="xxx">
       ...
     </group-specification>
  </structural-specification>
  ...
</organisational-specification>
```

# Structural specification – Role

Programming Organization

- role-definitions section:
  - identifier of the role (id attribute of role tag) with possible *inherited* roles (extends tag) - by default, all roles inherit of the soc role -

## Example

```
<role-definitions>
  <role id="player" />xs
  <role id="coach" />
  <role id="middle"> <extends role="player"/> </role>
  <role id="leader"> <extends role="player"/> </role>
  <role id="r1>
    <extends role="r2" />
    <extends role="r3" />
  </role>
  ...
</role-definitions>
```

# Structural specification – Group

Programming Organization

- group-specification section:
  - group identifier (id attribute of group-specification tag)
  - roles participating to this group and their cardinality (roles tag and id, min, max), i.e. min. and max. number of agents that should adopt the role in the group (default is 0 and unlimited)
  - links between roles of the group (link tag)
  - subgroups and their cardinality (subgroups tag)
  - formation constraints on the components of the group (formation-constraints)

## Example

```
<group-specification id="team">
   <roles>
        <role id="coach" min="1" max="2"/> ...
   </roles>
   <links> ... </links>
   <subgroups> ... </subgroups>
   <formation-constraints> ... </formation-constraints>
</group-specification>
```

# Functional specification

Programming Organization

- functional-specification section:
  - sequence of the schemes participating to the expected behaviour of the organisation

## Example

```
<functional-specification>
    <scheme id="sideAttack" >
        <goal id="dogoal" > ... </goal>
        <mission id="m1" min="1" max="5">
            ...
        </mission>
        ...
    </scheme>
    ...
</functional-specification>
```

# Functional specification – Scheme, Goal, Plan

Programming Organization

- Scheme definition (scheme tag) is composed of:
  - identifier of the scheme (id attribute of scheme tag)
  - the root goal of the scheme with the plan aiming at achieving it (goal tag)
  - the set of missions structuring the scheme (mission tag)
- Goal definition within a scheme (goal tag) is composed of:
  - an idenfier (id attribute of goal tag)
  - a type (achievement default or maintenance)
  - min. number of agents that must satisfy it (min) (default is “all”)
  - optionally, an argument (argument tag) that must be assigned to a value when the scheme is created
  - optionally a plan
- Plan definition attached to a goal (plan tag) is composed of
  - one and only one operator (operator attribute of plan tag) with sequence, choice, parallel as possible values
  - set of goal definitions (goal tag ) concerned by the operator

# Scheme – example

Programming Organization

```
<scheme id="sideAttack">
 <goal id="scoreGoal" min="1" >
  <plan operator="sequence">
    <goal id="g1" min="1" ds="get the ball" />
    <goal id="g2" min="3" ds="to be well placed">
      <plan operator="parallel">
        <goal id="g7" min="1" ds="go toward the opponent's field" />
        <goal id="g8" min="1" ds="be placed in the middle field" />
        <goal id="g9" min="1" ds="be placed in the opponent's goal area" />
      </plan>
    </goal>
    <goal id="g3" min="1" ds="kick the ball to the m2Ag" >
       <argument id="M2Ag" />
    </goal>
    <goal id="g4"       min="1" ds="go to the opponent's back line" />
    <goal id="g5"       min="1" ds="kick the ball to the goal area" />
    <goal id="g6"       min="1" ds="shot at the opponent's goal" />
  </plan>
 </goal>
 ...
```

# Functional specification – Mission

Programming Organization

- Mission definition (mission tag) in the context of a scheme definition, is composed of:
  - identifier of the mission (id attribute of mission tag)
  - cardinality of the mission min (0 is default), max (unlimited is default) specifying the number of agents that can be committed to the mission
  - the set of goal identifiers (goal tag) that belong to the mission

## Example

```
<scheme id="sideAttack">
  ... the goals ...
 <mission id="m1" min="1" max="1">
    <goal id="scoreGoal" /> <goal id="g1" />
    <goal id="g3" /> ...
  </mission>
  ...
</scheme>
```

# Normative specification

Programming Organization

- Explicit relation between the functional and structural specifications
- Permissions and obligations to commit to missions in the context of a role
- The normative specification makes explicit the normative dimension of a role
- normative-specification section:
  - sequence of the norm specifications participating to the governance of the organisation

## Example

```
<normative-specification>
    <norm id="n1" ... />
     ...
    <norm id="..." ... />
</normative-specification>
```

# Normative specification – Norm



- Norm definition (norm tag) in the context of a normative-specification definition, is composed of:
  - the identifier of the norm (id)
  - the type of the norm (type) with obligation, permission as possible values
  - optionally a condition of activation (condition) with the following possible expressions:
    - checking of properties of the organisation (e.g. #role_compatibility, #mission_cardinality, #role_cardinality, #goal_non_compliance)
    - ⇝ unregimentation of organisation properties !!!
    - (un)fulfillment of an obligation stated in a particular norm (unfulfilled, fulfilled)
  - the identifier of the role (role) on which the role is applied
  - the identifier of the mission (mission) concerned by the norm
  - optionally a time constraint (time-constraint)

# Norm – example



- Any agent playing *back* is *obliged* to commit to mission *m*1 and achieve its goals within 1 minute

```
<norm id = "n1" type="obligation"
      role="back" mission="m1" time-constraint="1 minute"/>
```

- Any agent playing *left* is *obliged* to commit to mission *m*2 and achieve its goals within 1 day

```
<norm id = "n2" type="obligation"
      role="left" mission="m2" time-constraint="1 day"/>
```

- Any agent playing *coach* is *obliged* to commit to mission *ms* and achieve its goals within 3 hour in case obligation of norm n2 has not been fulfilled

```
<norm id = "n4" type="obligation"
      condition="unfulfilled(obligation(_,n2,_,_))"
      role="coach" mission="ms" time-constraint="3 hour"/>
```

# Normative specification – NPL

Programming Organization

Norms can be written in Normative Programming Language (NPL):

- an activation condition
- a consequence
  - regimentations (`fail`)
  - obligations (`obligation`)
- terms starting with an upper case letter are variables

## Example (Norm)

```
norm n1: plays(A,writer,G) -> fail.
```

or

```
norm n1: plays(A,writer,G)
      -> obligation(A,n1,plays(A,editor,G),
         `now + 3 min`).
```

# Normative specification – NPL

Programming Organization

## Example (NPL Program)

```
<npl-norms>
     a :- t &amp;amp; k.
     norm npl1: a &amp;amp; v(X) ->
          obligation(bob,true,g(X),`now`+`1 day`).
     norm npl2: a &amp;amp; b    -> fail(test).
</npl-norms>
```

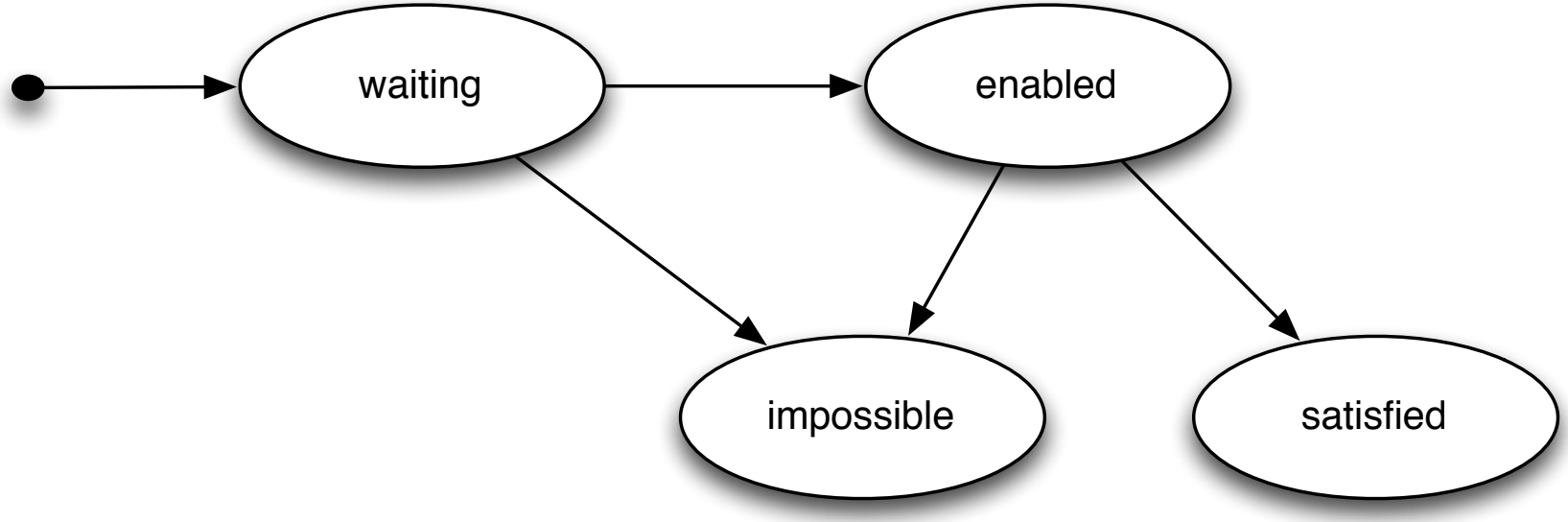
# Organisation entity dynamics



1. Organisation is created (by the agents)
   - instances of groups
   - instances of schemes
2. Agents enter into groups *adopting* roles
3. When a group is well formed, it may become *responsible* for schemes
   - Agents from the group are then obliged to commit to missions in the scheme
4. Agents *commit* to missions
5. Agents *fulfil* mission's goals
6. Agents leave schemes and groups
7. Schemes and groups instances are destroyed

# Goal dynamics

Programming Organization

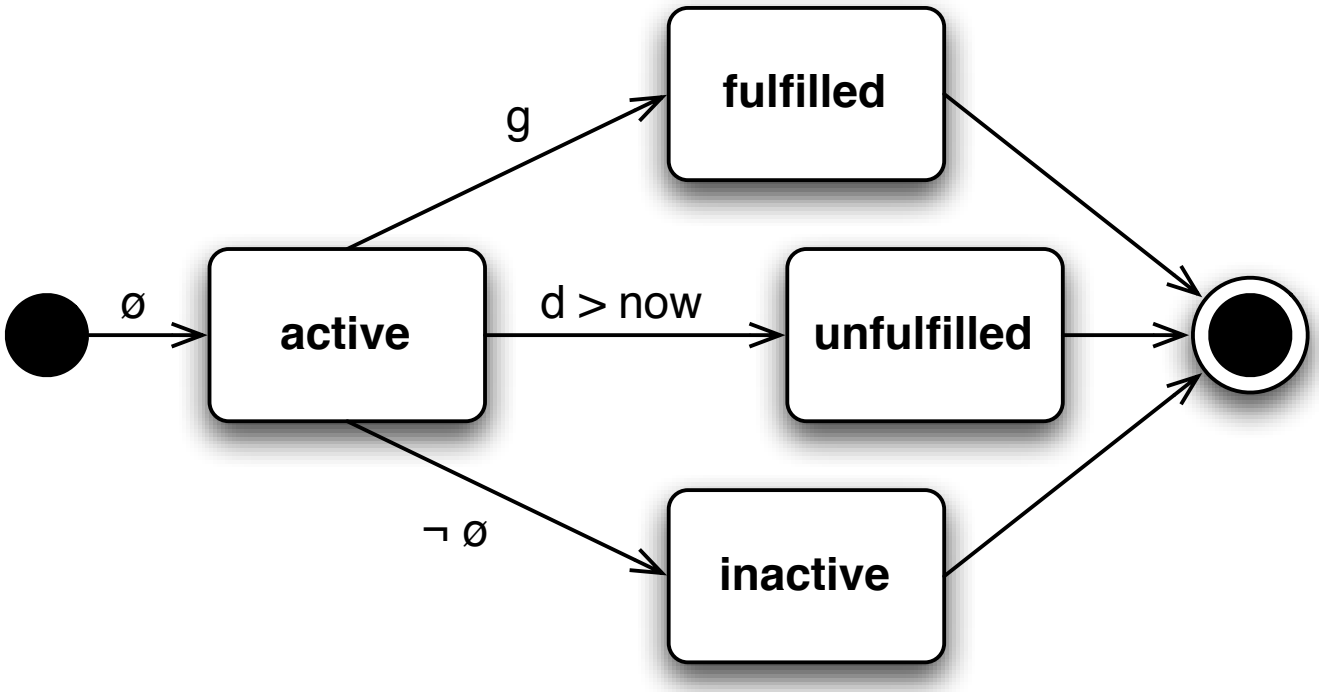



| | |
|---|---|
| waiting | initial state |
| enabled | goal pre-conditions are satisfied & scheme is well-formed |
| satisfied | agents committed to the goal have achieved it |
| impossible | the goal is impossible to be satisfied |

Note: goal state from the Organization point of view may be different of the goal state from the Agent point of view

# Norm dynamics

Programming Organization

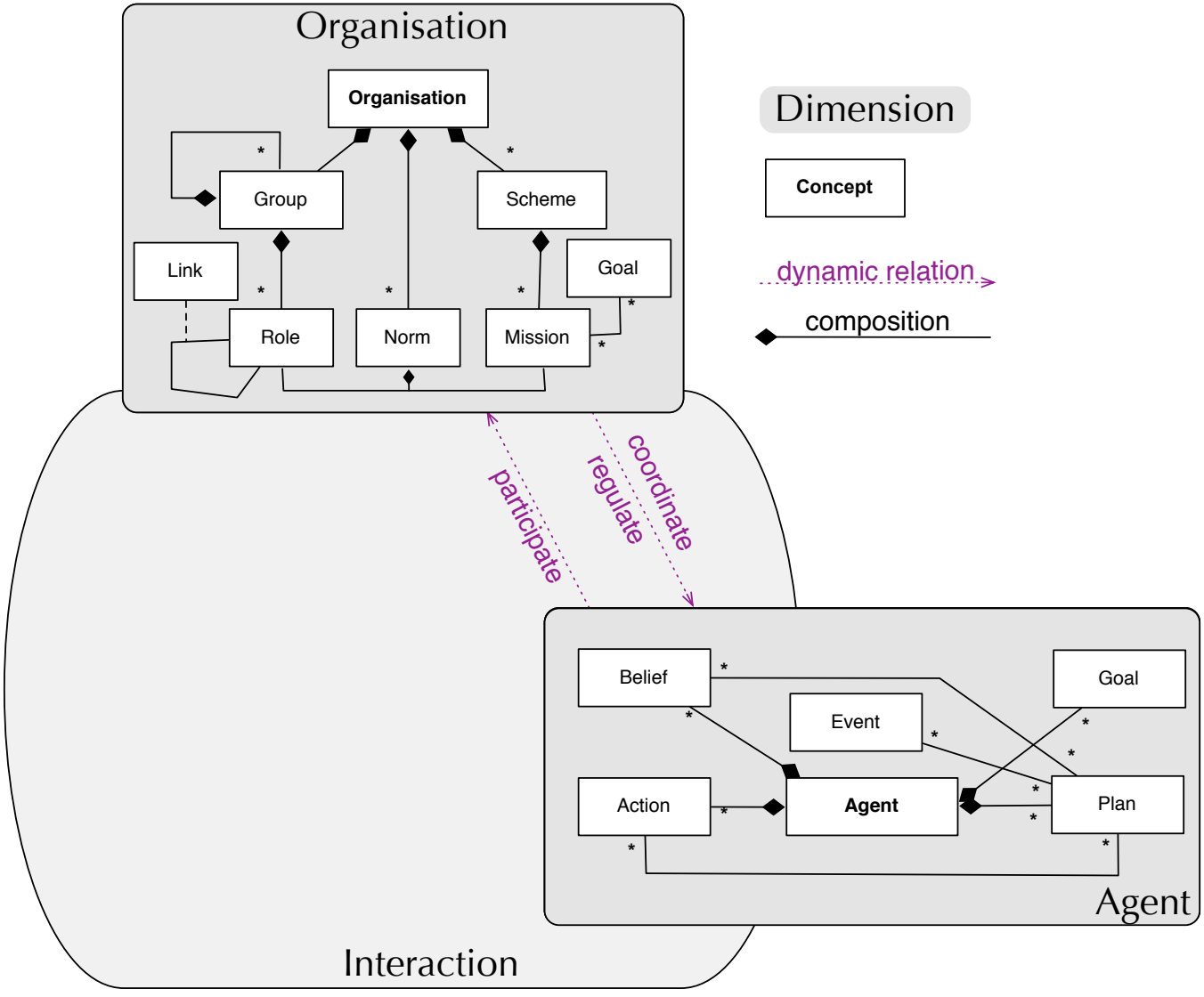



$$norm\ n: \ \phi -> \ obligation(a,r,g,d)$$

- $\phi$: activation condition of the norm (e.g. play a role)
- $g$: the goal of the obligation (e.g. commit to a mission)
- $d$: the deadline of the obligation

# Integrating A & O dimensions

Deeper walk in JaCaMo meta-model



# Programming Organization

*Agent* integration mechanisms allow agents to be aware of and to deliberate on:

- entering/exiting the organisation
- modification of the organisation
- obedience/violation of norms
- sanctioning/rewarding other agents

e.g. $\mathcal{J}$-$\mathcal{M}$OISE$^+$ [Hübner et al., 2007], Autonomy based reasoning [Carabelea, 2007], *ProsA*$_2$ Agent-based reasoning on norms [Ossowski, 1999], ...

# Organization actions and beliefs

Deeper walk in JaCaMo meta-model

- *Observable Properties*:
  - group(group_id,group_type,artid): list of the group_id of group_type that exist in the organizational entity
  - scheme(scheme_id,scheme_type,artid): list of the scheme_id of scheme_type that exist in the organizational entity
- *Operations*:
  - createGroup(group) (resp. removeGroup(grid)): attempts to create (resp. remove) group in the organization
  - createScheme(scheme) (resp. removeScheme(schid)): attempts to create (resp. remove) scheme in the organization

Note: available through OrgBoard Artifact created when creating an organization

# Group actions and beliefs

Deeper walk in JaCaMo meta-model

- *Observable Properties*:
  - specification: group spec. in the OS
  - player: list of play(agent, role, group)
  - schemes: list of scheme identifiers that the group is responsible for
  - subgroups, parentGroup, formationStatus (if the group is well formed or not)
- *Operations*:
  - adoptRole(role) (resp. leaveRole(role)): attempts to adopt (resp. leave) role in the group
  - addScheme(schid) (resp. removeScheme(schid)): attempts to set (resp. unset) the group responsible for the scheme managed by the SchemeBoard schId
  - setParentGroup(groupid), setOwner(agtid), destroy

Note: available through GroupBoard Artifact created when creating a group in an organization

# Scheme actions and beliefs

Deeper walk in JaCaMo meta-model

- *Observable Properties*:
  - specification: scheme spec. in the OS
  - commitments: list of commitment(agent, mission, scheme)
  - groups: list of groups resp. for the scheme
  - goalState: list of goals' current state
  - goalArgument(schemeId,goalId,argId,value): added only if the argument has a value, usually defined by the operation setArgumentValue
  - obligations: list of active obligations in the scheme (obligation(agt,norm,goal,deadline))
  - permissions: list of active permissions in the scheme (permission(agt,norm,goal,deadline))
  - goalArgument: value of goals' arguments, defined by the operation setArgumentValue
- *Operations*:
  - commitMission(mission) (resp. leaveMission): attempts to "commit" (resp "leave") a mission in the scheme
  - goalAchieved(goal): declares that goal is achieved
  - setArgumentValue(goal, argument, value): defines the value of goal's argument
  - resetGoal(goal) (reset the status of a goal), destroy

Note: available in SchemeBoard Artifact created when creating a scheme in an organization

# Norm actions and beliefs

Deeper walk in JaCaMo meta-model

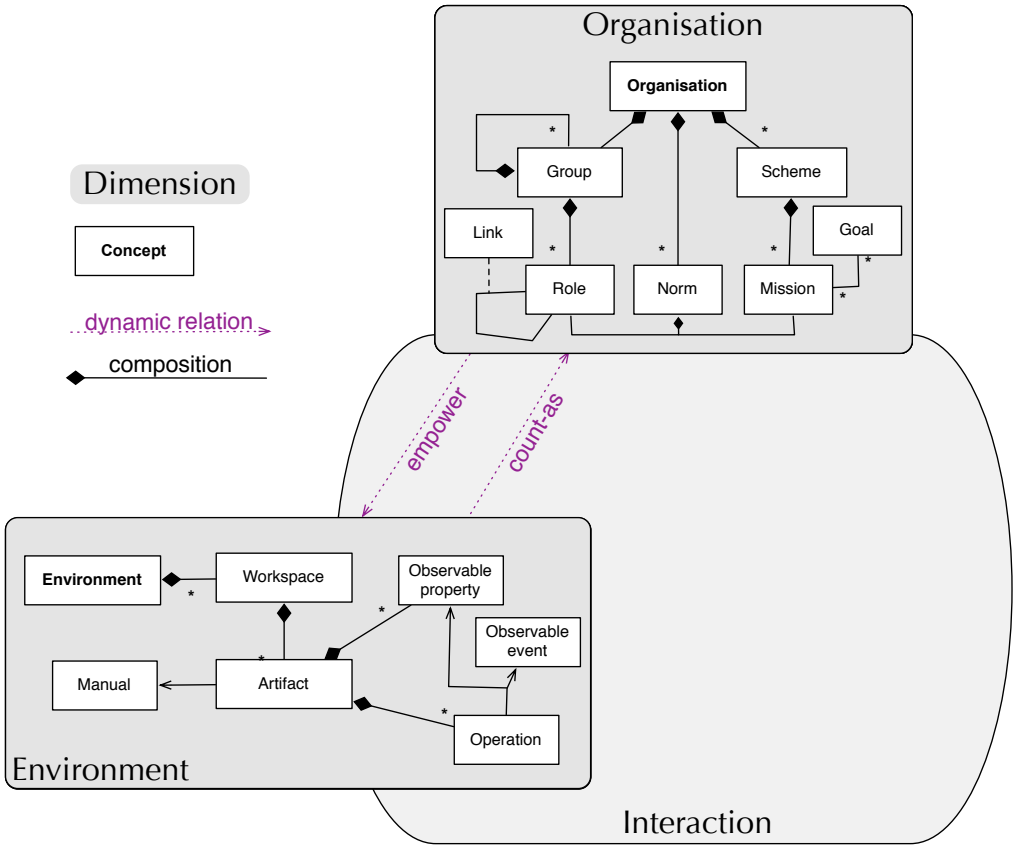
- *Observable Properties*:
  - obligation: current active obligations
- *Operations*:
  - load(nplprogram)
  - addFact (resp. removeFact)

Note: available in Normative board managing obligations/permissions defined in the normative specification

- automatically created when a group becomes responsible for a scheme
- or when loading any NPL program

# Integrating O & E dimensions

Deeper walk in JaCaMo meta-model




Transforming organisations into embodied organisations [de Brito et al., 2012], [Piunti et al., 2009], [Okuyama et al., 2008] so that:

- organisation may act on the environment (e.g. enact rules, regimentation)
- environment may act on the organisation (e.g. count-as rules) based on Situated Artificial Institution [de Brito et al., 2015]

# Outline

Programming Agents

Programming Environment

Programming Organization

Advanced practice of JaCaMo

Conclusion and Perspective

# Advanced practice of JaCaMo

- Playing with Dimensions of Coordination (available in doc/tutorials/coordination)
  - Illustration of different approaches to coordinate agents in JaCaMo taking profit of the three programming dimensions of the platform, that is to say:
    - coordinating by focusing on agent (e.g. direct message passing),
    - coordinating by focusing on the environment (e.g. coordination artifact),
    - coordinating by focusing on the organisation (e.g. coordination strategies within organisation specification)
- BDI Hello World
  - Illustration of how the BDI model is used in the JaCaMo Agent programming language exploring the BDI features of Jason.
- Gold Miners (available also in doc/tutorials/gold-miners)
  - Illustration of some of the features of the JaCaMo Agent Programming Language through the development and refinement of the Gold Miners contest scenario implementation
- Building House (available also in doc/tutorials/house-building)
  - Illustration of using JaCaMo for negotiating contracts and monitoring their execution

# Outline

Programming Agents

Programming Environment

Programming Organization

Advanced practice of JaCaMo

Conclusion and Perspective

# Conclusions

- MAS is not only agents
- MAS is not only organisation
- MAS is not only environment
- MAS is not only interaction

MAS has many dimensions
All as first class entities

# Conclusions

Multi-Agent Oriented Programming proposes a seamless integration of different abstractions that brings interesting features to program collective autonomous systems:

- *Separation of concerns* between Agent, Environment, Organisation and Interaction
- *Openness and heterogeneity*
- *Flexibility, adaptation, rich coordination and regulation driven by agents, environment, interactions or organisations*
- *Inclusion of Physical, Digital and Human worlds to define socio-cognitive, physical and digital systems*
- *Programming features*: Modularity, extensibility, reusability, readability, ...

# Some open issues & Perspectives

Engineering perspective:

- Debugging, Performance, ...
- Life cycle of MAS (from requirement to maintenance) $\rightsquigarrow$ software engineering tools and methods
- Shift from Agent-Oriented Sofware Engineering to Multi-Agent Oriented Software Engineering where all the dimensions A, E, I, O may guide each step of the process (cf. [Uez and Hübner, 2014])
- Evaluation & Verification of MAO programmed applications,
- Integrating with other technologies
- Handle Scalability, Robustness

# Multi-Agent Oriented Programming


O. Boissier

Univ. Lyon, IMT Mines Saint-Etienne, LaHC UMR CNRS 5516, France

in tight collaboration with R.H. Bordini[2], J.F. Hübner[3], A. Ricci[4]

[2] Pontificia Universidade Catolica do Rio Grande do Sul (PUCRS), Porto Alegre, Brazil
[3] Federal University of Santa Catarina (UFSC), Florianópolis, Brazil
[4] University of Bologna (UNIBO), Bologna, Italy


Winter 2019



# Bibliography I


Boissier, O., Bordini, R. H., Hübner, J. F., Ricci, A., and Santi, A. (2011).

Multi-agent oriented programming with jacamo.

*Science of Computer Programming*, pages –.

Bordini, R. H., Hübner, J. F., and Wooldrige, M. (2007).

*Programming Multi-Agent Systems in AgentSpeak using Jason*.

Wiley Series in Agent Technology. John Wiley & Sons.

Carabelea, C. (2007).

*Reasoning about autonomy in open multi-agent systems - an approach based on the social power theory*.

in french, ENS Mines Saint-Etienne.

de Brito, M., Hübner, J. F., and Boissier, O. (2015).

Bringing constitutive dynamics to situated artificial institutions.

In *Proc. of 17th Portuguese Conference on Artificial Intelligence (EPIA 2015)*, volume 9273 of *LNCS*, pages 624–637. Springer.

de Brito, M., Hübner, J. F., and Bordini, R. H. (2012).

Programming institutional facts in multi-agent systems.

In *COIN-12, Proceedings*.

# Bibliography II


Hübner, J. F., Boissier, O., Kitio, R., and Ricci, A. (2009).

Instrumenting Multi-Agent Organisations with Organisational Artifacts and Agents.

*Journal of Autonomous Agents and Multi-Agent Systems*.

Hübner, J. F., Sichman, J. S., and Boissier, O. (2007).

Developing Organised Multi-Agent Systems Using the MOISE+ Model: Programming Issues at the System and Agent Levels.

*Agent-Oriented Software Engineering*, 1(3/4):370–395.

Okuyama, F. Y., Bordini, R. H., and da Rocha Costa, A. C. (2008).

A distributed normative infrastructure for situated multi-agent organisations.

In Baldoni, M., Son, T. C., van Riemsdijk, M. B., and Winikoff, M., editors, *DALT*, volume 5397 of *Lecture Notes in Computer Science*, pages 29–46. Springer.

Omicini, A., Ricci, A., and Viroli, M. (2008).

Artifacts in the A&A meta-model for multi-agent systems.

*Autonomous Agents and Multi-Agent Systems*, 17(3):432–456.

# Bibliography III


Ossowski, S. (1999).

*Co-ordination in Artificial Agent Societies: Social Structures and Its Implications for Autonomous Problem-Solving Agents*, volume 1535 of *LNAI*.

Springer.

Piunti, M., Ricci, A., Boissier, O., and Hübner, J. F. (2009).

Embodied organisations in mas environments.

In Braubach, L., van der Hoek, W., Petta, P., and Pokahr, A., editors, *Proceedings of 7th German conference on Multi-Agent System Technologies (MATES 09), Hamburg, Germany, September 9-11*, volume 5774 of *LNCS*, pages 115–127. Springer.

Ricci, A., Piunti, M., Viroli, M., and Omicini, A. (2009).

Environment programming in CArtAgO.

In *Multi-Agent Programming: Languages,Platforms and Applications,Vol.2*. Springer.

# Bibliography IV

Uez, D. M. and Hübner, J. F. (2014).

Environments and organizations in multi-agent systems: From modelling to code.

In Dalpiaz, F., Dix, J., and van Riemsdijk, M. B., editors, *Engineering Multi-Agent Systems - Second International Workshop, EMAS 2014, Paris, France, May 5-6, 2014, Revised Selected Papers*, volume 8758 of *Lecture Notes in Computer Science*, pages 181–203. Springer.