Agent-Oriented Programming Tutorial

Leuven October 2011

9-2-2012

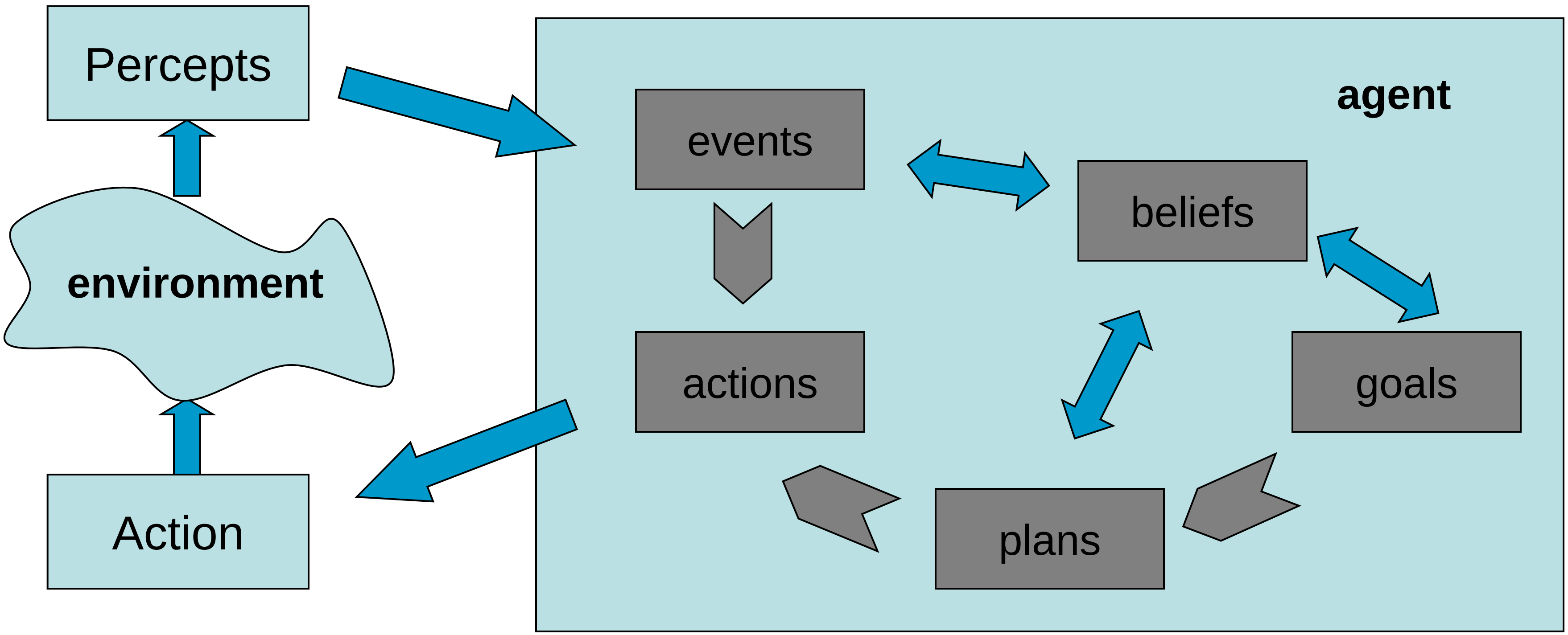
Overview

- 13u 'Half-day tutorial' on AOPLs/GOAL
- 13u Overview of AOPLs, Environment interaction
- 14u30 Coffee break
- 15u GOAL Tutorial
- 18u30 End

Topics today

- 1. Agent Programming Languages
- 2. JACK Agent Language
- 3. Architectures & Reasoning Cycles
- 4. Environment Interfacing
- 5. Research Themes
- 6. BDI: Goals
- 7. GOAL: Goal-Oriented Agent Language
- 8. Action Specification & Selection
- 9. Sensing & Environments
- 10. Instantaneous & Durative Actions
- 11. GOAL: Program Structure
- 12. Modularity & Multiple Agents

Agents: Represent environment



Agent Oriented Programming

- **Agents** provide a very effective way of building applications for **dynamic and complex environments**

+

- Develop **agents** based on **Belief-Desire-Intention agent metaphor**, i.e. develop software components <u>as if</u> they have **beliefs** and **goals**, **act** to achieve these goals, and are able to **interact** with their environment and other agents.

1.

Agent Programming Languages

A Brief History of AOP

- 1990: AGENT-0 (Shoham)
- 1993: PLACA (Thomas; AGENT-0 extension with plans)
- 1996: AgentSpeak(L) (Rao; inspired by PRS)
- 1996: Golog (Reiter, Levesque, Lesperance)
- 1997: 3APL (Hindriks et al.)
- 1998: ConGolog (Giacomo, Levesque, Lesperance)
- 2000: JACK (Busetta, Howden, Ronnquist, Hodgson)
- 2000: GOAL (Hindriks et al.)
- 2000: CLAIM (Amal El FallahSeghrouchni)
- 2002: Jason (Bordini, Hubner; implementation of AgentSpeak)
- 2003: Jadex (Braubach, Pokahr, Lamersdorf)
- 2008: 2APL (successor of 3APL)

This overview is far from complete!

A Brief History of AOP

• AGENT-0	Speech acts
• PLACA	Plans
• AgentSpeak(L)	Events/Intentions
• Golog	Action theories, logical specification
• 3APL	Practical reasoning rules
• JACK	Capabilities, Java-based
• GOAL	Declarative goals
• CLAIM	Mobile agents (within agent community)
• Jason	AgentSpeak + Communication
• Jadex	JADE + BDI
• 2APL	Modules, PG-rules, …

A Brief History of AOP

Agent Programming Languages and Agent Logics have not (yet) converged to a uniform conception of (rational) agents.

Agent Programming	Agent Logics
Architectures PRS (Planning) , InterRap	BDI, Intention Logic, KARO
Agent-Oriented Programming Agent0, AgentSpeak, ConGolog, 3APL/2APL, Jason, Jadex, JACK, …	Multi-Agent Logics, Norms, Collective Intentionality
Conceptual extension “Declarative Goals”	CASL, Games and Knowledge

Agent Features

Many diverse and different features have been proposed, but the unifying theme still is the BDI view of agents.

Agent Programming	Agent Logics
• "Simple" beliefs and belief revision • Planning and Plan revision e.g. Plan failure • Declarative Goals • Triggers, Events e.g. maintenance goals • Control Structures • …	• "Complex" beliefs and belief revision • Commitment Strategies • Goal Dynamics • Look ahead features e.g. beliefs about the future, strong commitment preconditions • Norms • …

How are these APLs related?

A comparison from a high-level, conceptual point, not taking into account any practical aspects (IDE, available docs, speed, applications, etc)

Family of Languages
Basic concepts: beliefs, action, plans, goals-to-do):

AgentSpeak(L), Jason[1]

AGENT-0[1] (PLACA) - - - // \\

Golog = 3APL[2]

Multi-Agent Systems
All of these languages (except AGENT-0, PLACA, JACK) have versions implemented "on top of" JADE.

Main addition: Declarative goals

2APL ≈ 3APL + GOAL

Java-based BDI Languages

Jack (commercial), Jadex

Mobile Agents

CLAIM[3]

[1] mainly interesting from a historical point of view
[2] from a conceptual point of view, we identify AgentSpeak(L) and Jason
[3] without practical reasoning rules
[4] another example not discussed here is AgentScape (Brazier et al.)

2.

JACK Agent Language

JACK Agent Language

Extends Java with …

Class Constructs

- Agent, Event, Plan, Capability, Beliefset, View

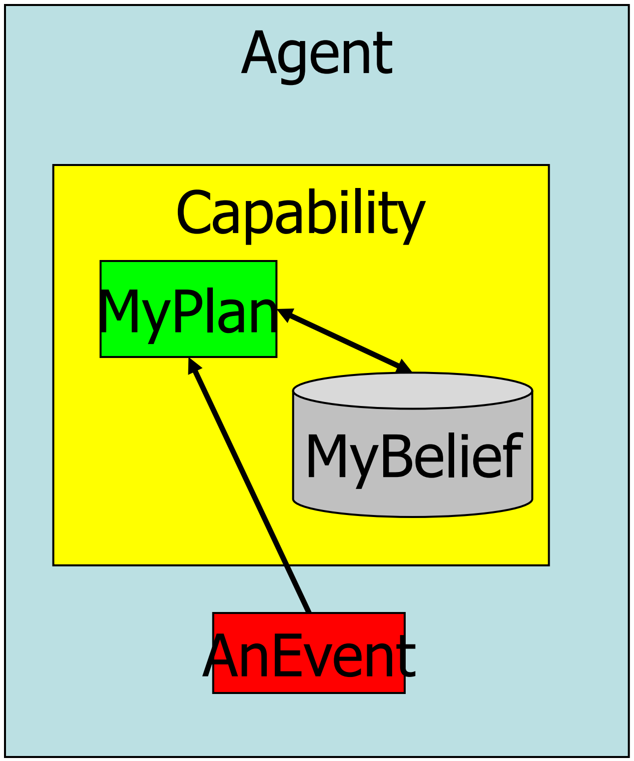
Declarations

- #handles, #uses, #posts, #sends, #reads, ...

Reasoning Method Statements ("at-statements")

- @wait-for, @maintain, @send, @reply, @subtask, @post, @achieve, @insist, @test, @determine

How do these pieces fit?




```
plan MyPlan extends Plan {
#handles event AnEvent ev;
#modifies data MyBelief b;
    context() { … }
    body() {
    // JACK code here
    // Java code can be used
    @post(…);
    }
}
```

Capabilities

- Encapsulates agent functionalities into “clusters”, i.e. modularity construct
- Represent functional aspects of an agent that can be “plugged in” as required
- Similar to agents, but:
  - can be nested (“sub-agents”), hence distinguish external/internal
  - don’t have constructors
  - don’t have identity (can’t send message to capability)
  - don’t have autonomy

Event

- Events trigger plans
- Provides the type safe connections between agents and plans:
  - both agents and plans must declare the events they handle as well as the events they post or send
- Range of types: Event, MessageEvent, BDIMessageEvent, BDIGoalEvent, …
  - MessageEvent: inter-agent
  - BDIGoalEvent: retry upon failure

Declaring & Posting Events

```
public event AddMeetingEvent extends Event {
    public Task task;
    #posted as newMeeting(Task task) {
      this.task = task;
    }
}
--------------------------------------------------
plan AddMeetingPlan extends Plan {
#handles ReqMeetingEvent reqamev;
#posts event AddMeetingEvent ev;
...
body(){
   ...
   @subtask(ev.newMeeting(reqamev.task));
   }
}
```

Plan Structure

plan *PlanName* extends **Plan** {

#handles event *EventType event_ref;*

// Plan method definitions and JACK Agent Language #-statements

// describing relationships to other components, reasoning methods, etc.

#posts event *EventType event_ref*;

#sends event *MessageEventType event_ref;*

#uses/reads/modifies data Type *ref/name;*

static boolean relevant (EventType *reference*) {

// code to test whether the plan is relevant to an event instance

}

context() { /* logical condition to test applicability */ }

body() {

// The plan body describing the actual steps performed when the

// plan is executed. Can contain Java code and @-statements.

}

/* Other reasoning methods here */

}

Summary

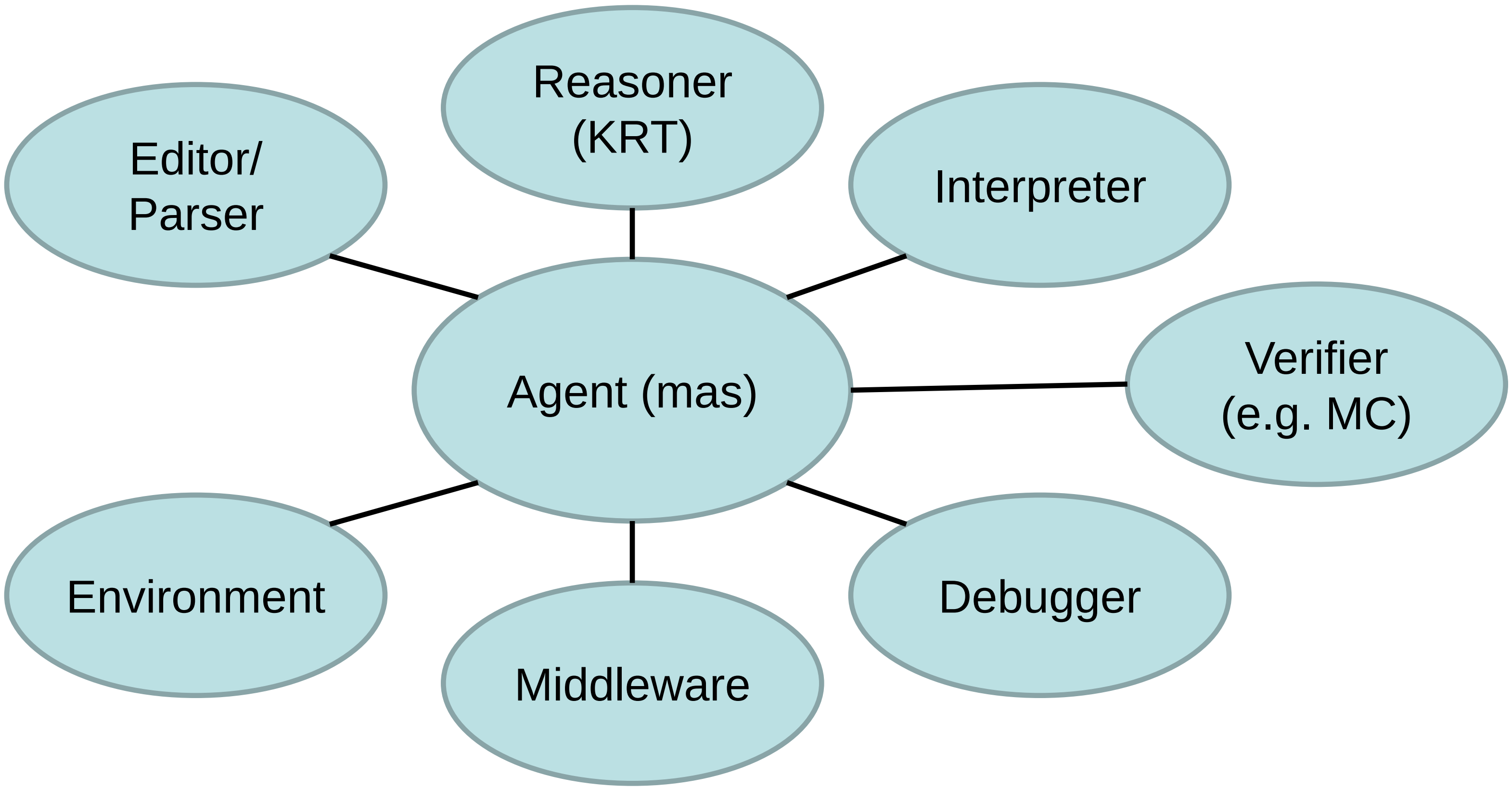
- JACK is a commercial agent platform/language aimed at industry
- JACK = Language + Platform + Tools
- JACK language extends Java with:
  - keywords (agent, event, plan, capability, belief, view)
  - #-declarations (#uses #sends #posts …)
  - @-statements (@achieve, @send, …)
- JACK provides various tools for building and debugging agent systems

3.

Architectures & Reasoning Cycles

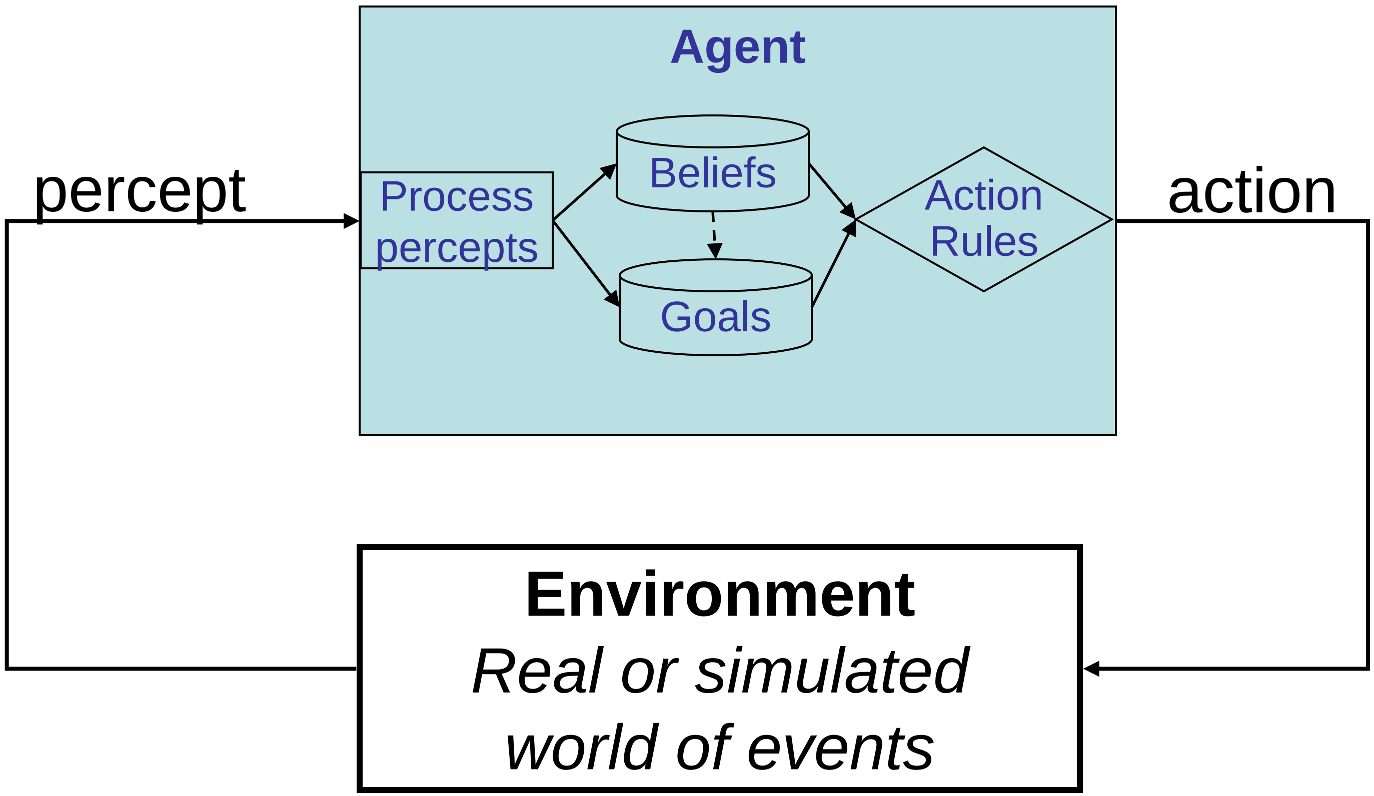
What is an Agent?

Structurally, an agent is a set of modules.
Infrastructurally, developing and running an agent requires of a set of components.

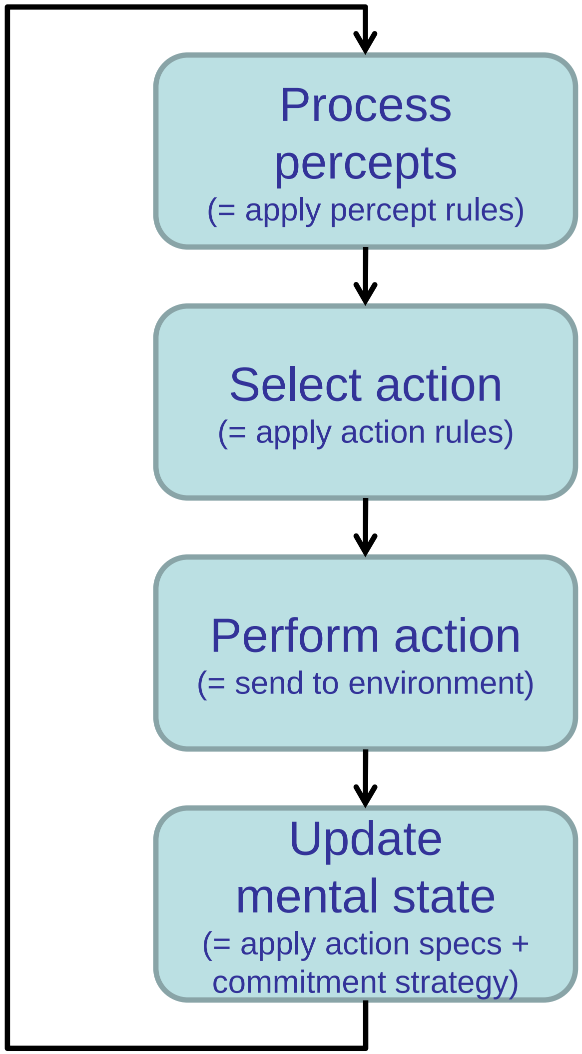


Towards standards: **EIS**, **KIS**, **MIS**. We also can define MCIS?
Parser, interpreter, and debugger are language/platform dependent?
Raises question: which agent platform features are used? Most useful?

GOAL Architecture



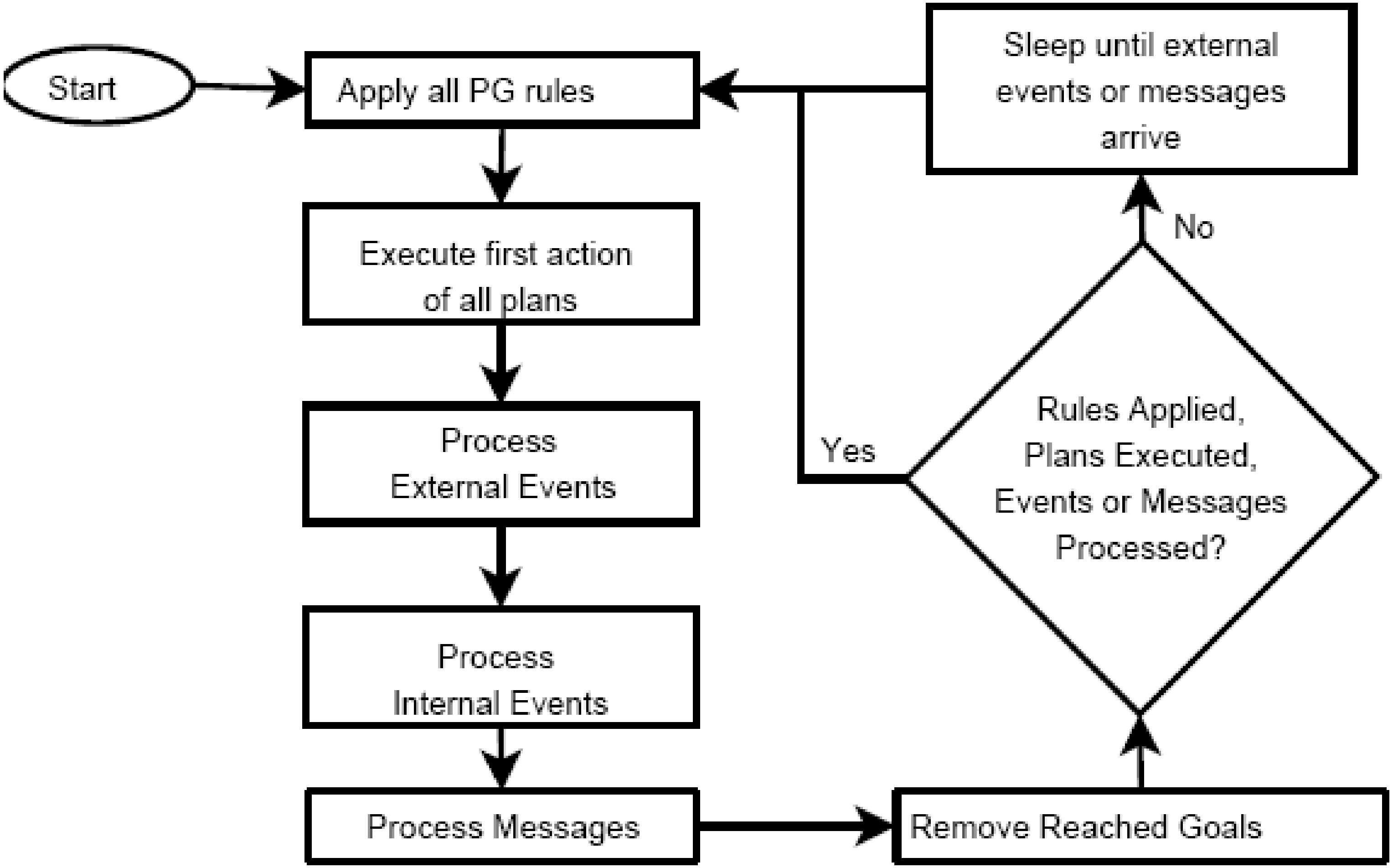
Interpreters: GOAL



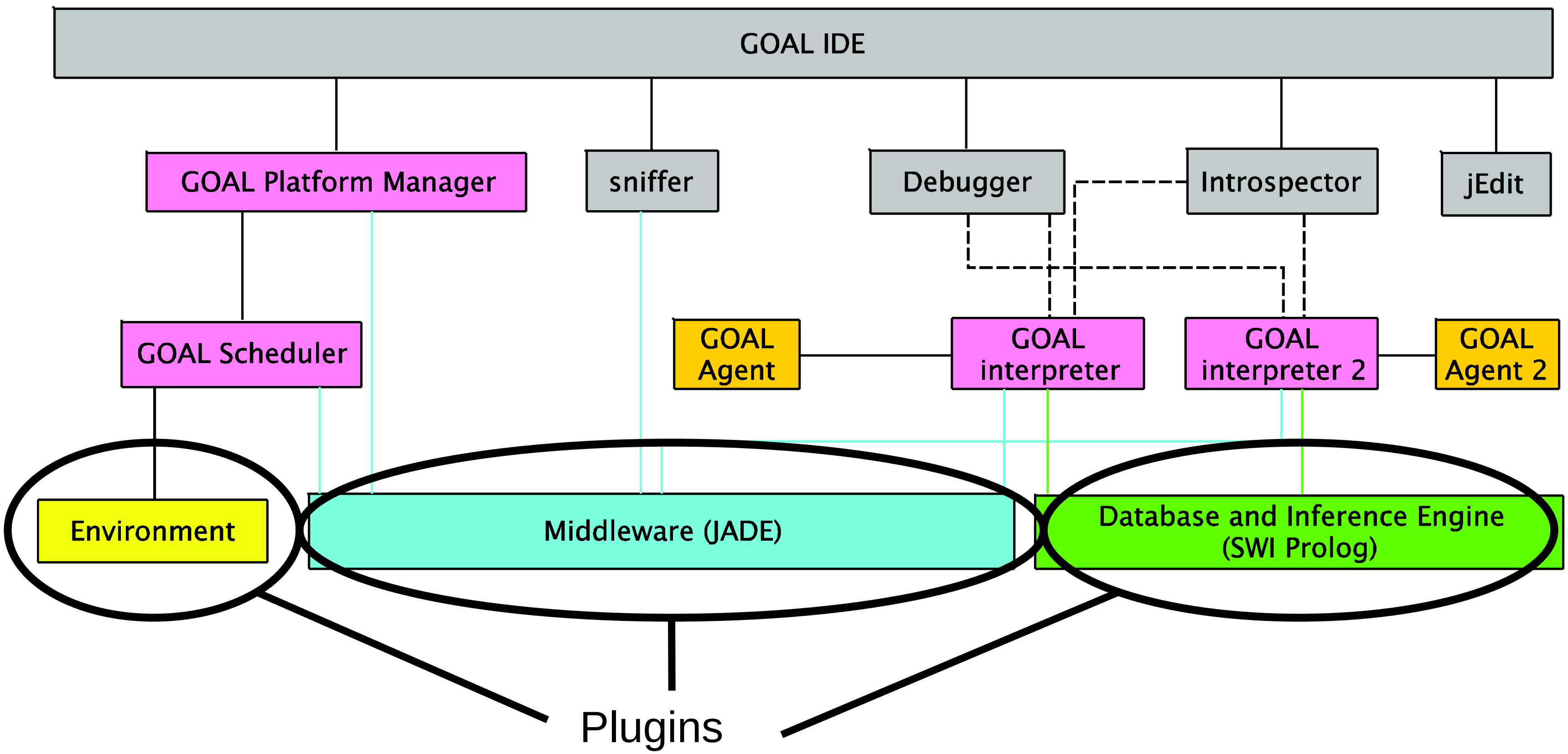
Also called **deliberation cycles**.

GOAL's cycle is a classic **sense-plan-act** cycle.

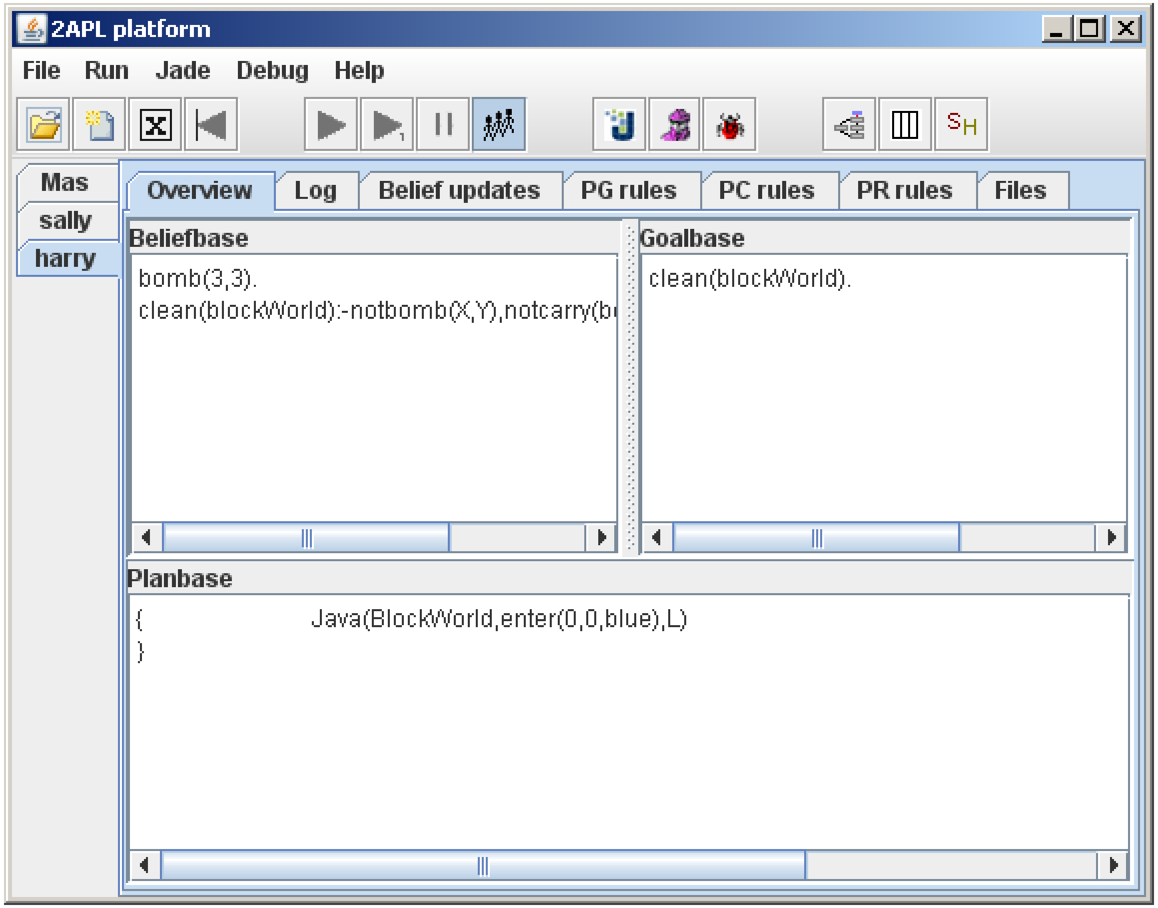
Interpreter: 2APL



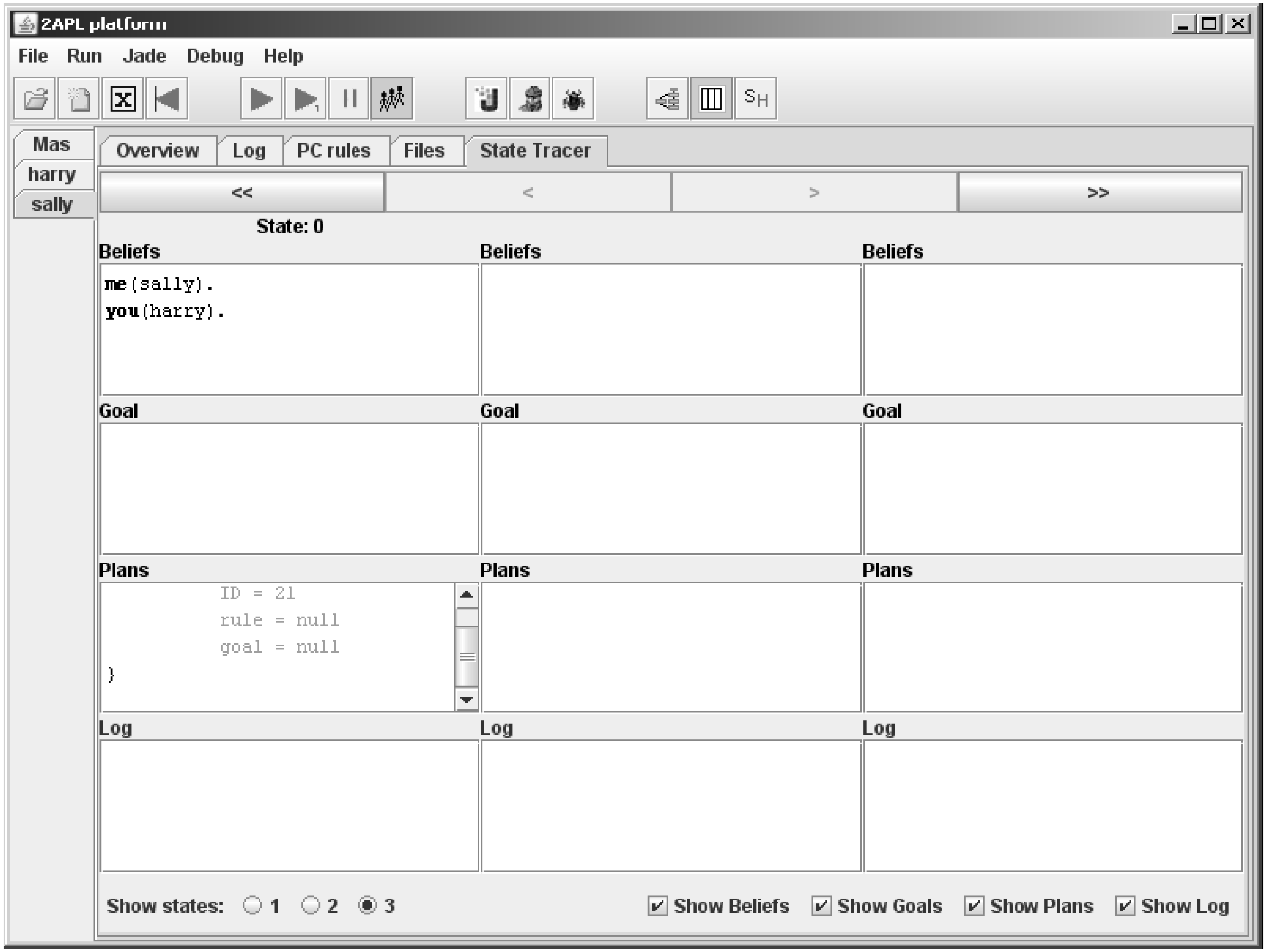
Under the Hood: Implementing AOP Example: GOAL Architecture



2APL IDE: Introspector



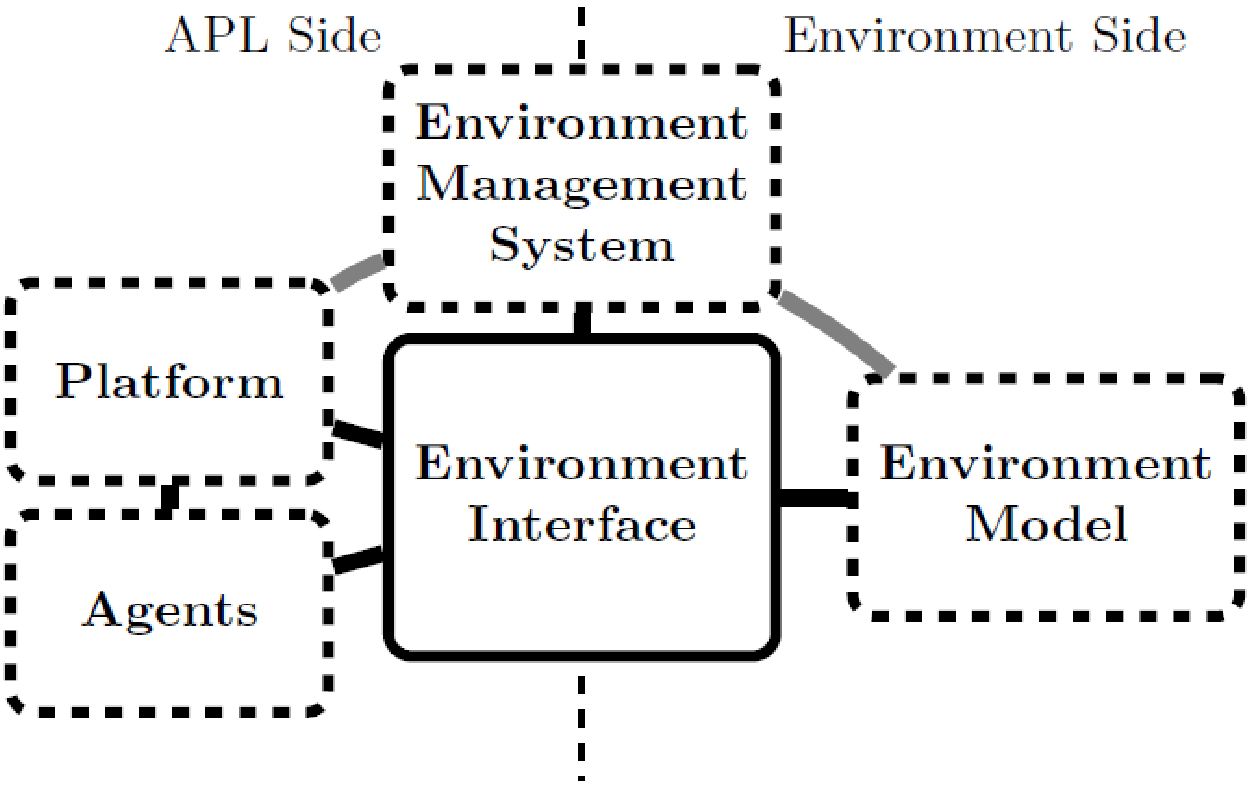
2APL IDE: State Tracer



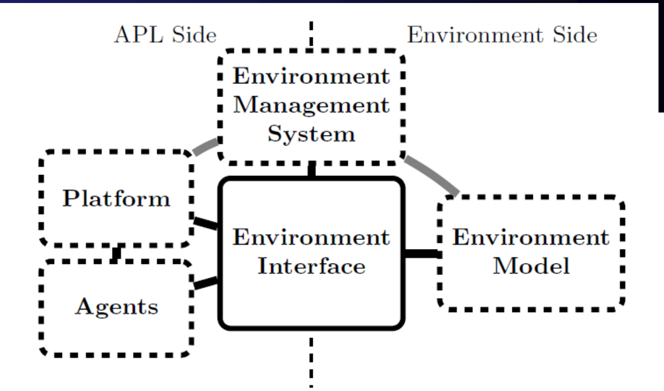
4.

Environment Interfacing
Environment Inteface Standard (EIS)

Design of a Generic Interface
Meta-Model



Design of a Generic Interface
Meta-Model



Controllable entities

- Entities in an environment that can be controlled by an agent
- Can be uniquely identified
- Provide sensors and actuators

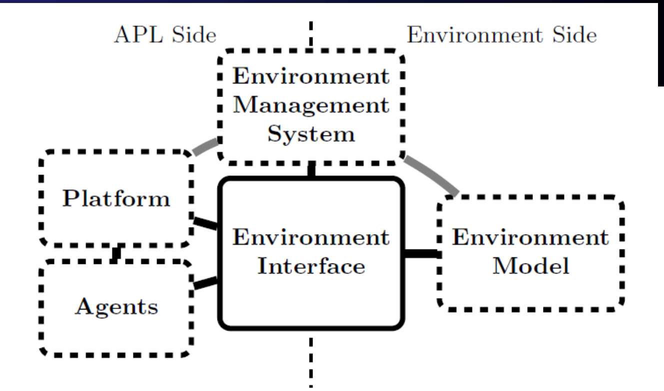
Agent

- anything that can be viewed as perceiving its environment through sensors and acting upon that environment through effectors (Russell & Norvig)

Environment interface

- Provides functionality for connecting agents to controllable entities
- Provides pull & notification-based mechanism for percepts
- Supports various action execution mechanisms

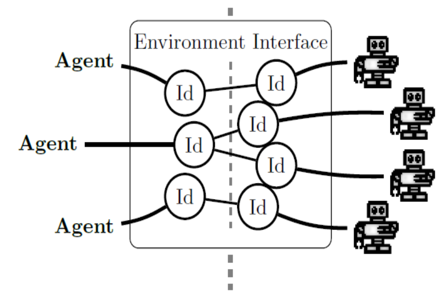
Design of a Generic Interface
Meta-Model



- Environment Management
  - initialize the environment using configuration files
  - release environment resources
  - kill environment or entity
  - pause
  - restart
  - reset

The Interface Functionality

1. attaching, detaching, and notifying observers
   – Facilitates notification of agent platforms
2. registering and unregistering age



3. adding and removing entities
4. managing the agents-entities relation
5. performing actions and retrieving percepts
6. managing the environment

Interface Intermediate Language

- convention for representing actions and percepts
- supports the exchange of percepts and actions from/to environments

- data containers: labels with arguments
- arguments can be identifiers, numerals, functions, lists

5.

Research Themes

A Research Agenda

Fundamental research questions:

- What kind of **expressiveness*** do we need in AOP? Or, what needs to be improved from your point of view? We need your feedback!
- **Verification**: Use e.g. temporal logic combined with belief and goal operators to prove agents "correct". Model-checking agents, mas(!)

Short-term important research questions:

- **Planning**: Combining reactive, autonomous agents and planning.
- **Learning**: How can we effectively integrate e.g. reinforcement learning into AOP to optimize action selection?
- **Debugging**: Develop tools to effectively debug agents, mas(!). Raises surprising issues: Do we need agents that revise their plans?
- **Scalability**: Develop efficient agent tools and interpreters that scale in practice.
- Last but not least, (your?) **applications**!

* e.g. maintenance goals, preferences, norms, teams, ...

Combining AOP and Planning

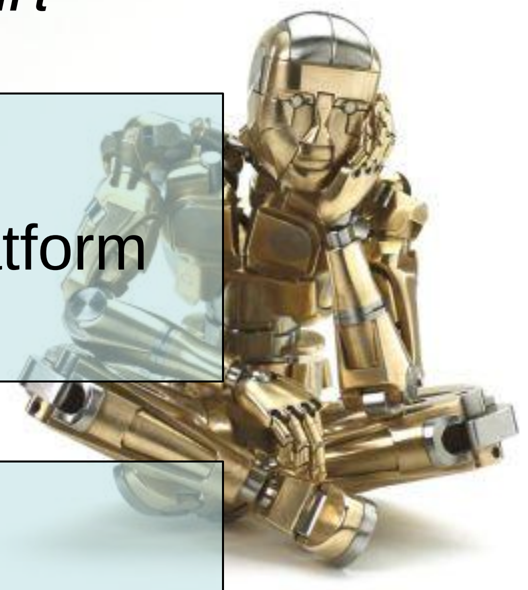
Combining the benefits of reactive, autonomous agents and planning algorithms

GOAL	Planning
• Knowledge	• Axioms
• Beliefs	• (Initial) state
• Goals	• Goal description
• Program Section	• x
• Action Specification	• Plan operators

Alternative KRT Plugin:
Restricted FOL, ADL, Plan Constraints (PDDL)

Applications

Need to apply the AOP to find out what works and what doesn't



- Use APLs for Programming Robotics Platform

- Many other possible applications:
- (Serious) Gaming (e.g. RPG, crisis management, …)
- Agent-Based Simulation
- The Web
- *<add your own example here>*

References

- 2APL: http://www.cs.uu.nl/2apl/
- ConGolog: http://www.cs.toronto.edu/cogrobo/main/systems/index.html
- GOAL: http://mmi.tudelft.nl/~koen/goal
- JACK: http://en.wikipedia.org/wiki/JACK_Intelligent_Agents
- Jadex: http://jadex.informatik.uni-hamburg.de/bin/view/About/Overview
- Jason: http://jason.sourceforge.net/JasonWebSite/Jason Home.php

- Multi-Agent Programming Languages, Platforms and Applications, Bordini, R.H.; Dastani, M.; Dix, J.; El Fallah Seghrouchni, A. (Eds.), 2005

  introduces 2APL, CLAIM, Jadex, Jason
- Multi-Agent Programming: Languages, Tools and Applications Bordini, R.H.; Dastani, M.; Dix, J.; El Fallah Seghrouchni, A. (Eds.), 2009

  introduces a.o.: Brahms, CArtAgO, GOAL, JIAC Agent Platform

6.

BDI: Goals

Achievement Goals

- Implemented cognitive agent programming languages typically incorporate achievement goals
- Achievement goal: goal to reach a certain state of affairs e.g., be at a certain location, have a weapon, have a clean floor, have a block on top of another block
- Declarative goal
- Different ways of representing goals
- Different semantics for goals

Jason – achievement goals (1)

http://jason.sourceforge.net/Jason/Jason.html

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

achievement goal (creation)

achievement goal (plan trigger)

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

+!at(Coords) ...
```

Jason - achievement goals (2)

- Represented as predicate !p(t1,...,tn)
- Used as plan triggers
- Created from within plans
- Stored as events in event base

Jadex – achievement goals

http://jadex.informatik.uni-hamburg.de/xwiki/bin/view/About/Overview

```
<goals>
    <achievegoal name="translate">
        <parameter name="direction" class="String"/>
        <parameter name="word" class="String"/>
        <parameter name="result" class="String" direction="out"/>
    </achievegoal>
</goals>
```

- Specified in XML
- Used as plan triggers
- Created from within plans in Java

```
IGoal goal = createGoal("translate");...;
dispatchSubgoalAndWait(goal);
```

- Stored as objects in goal base

GOAL - achievement goals

- Represented as conjunctions of atoms
  $p_1(t1,\ldots,tn), \ldots, p_k(t1,\ldots,tm)$
- Used for action selection
- Created from within action rules
- Stored in goal base

Commitment Strategy of GOAL

- Goals are dropped from goal base when believed to be achieved (deletion perspective)

  ≈ blind commitment

- Mental state condition `a-goal(φ)` holds if φ is not believed (satisfaction perspective)

- Goals can also be dropped using the built-in `drop` action
  - can be used to implement single-minded commitment

7.

GOAL: Goal-Oriented Agent Language

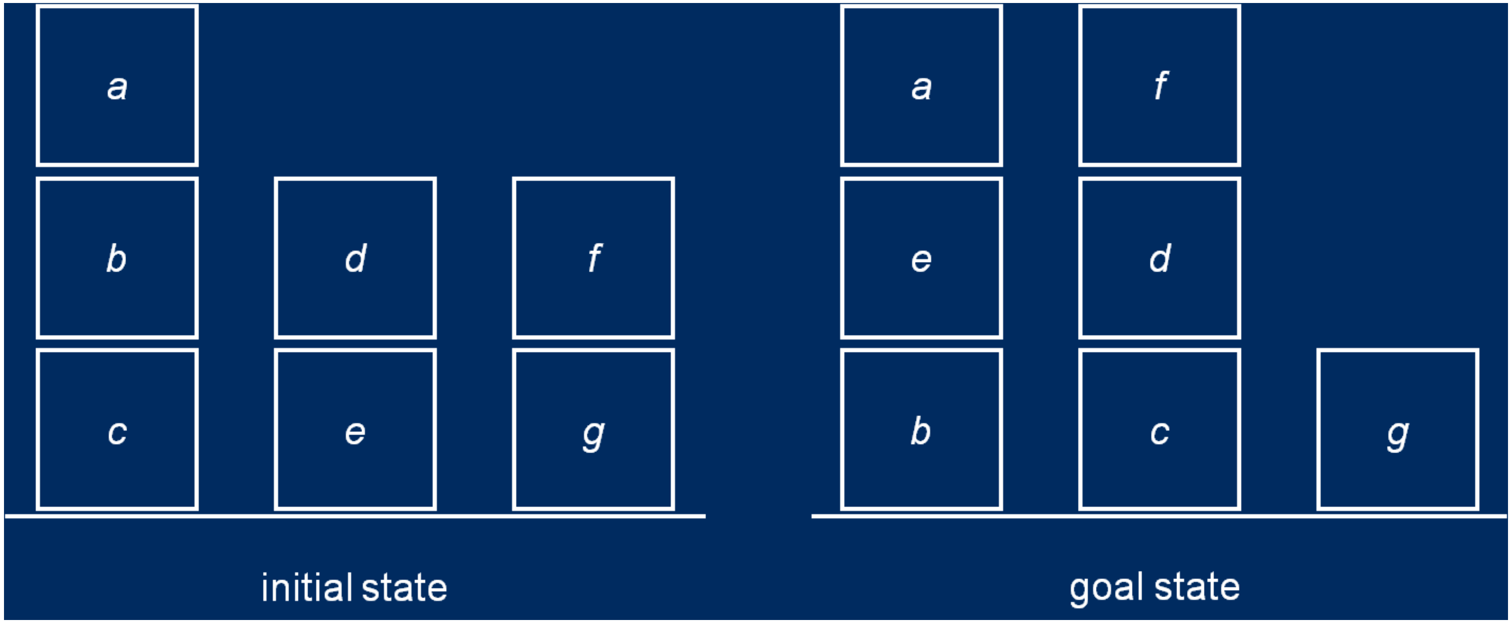
GOAL Mental State: Overview

- Beliefs
  represent current state of environment (Prolog)
- Knowledge
  represent (static) domain knowledge (Prolog)
- Goals
  represent achievement goals (conjunctions of atoms)

The Blocks World

A *classic AI* planning *problem.*

Objective: Move blocks in initial state such that result is goal state.



- *Positioning* of blocks on table is not relevant.
- A block can be moved *only if* it there is no other block on top of it.

Representing the Blocks World

***Prolog is the* knowledge representation language *used in* GOAL.**

Basic predicates:

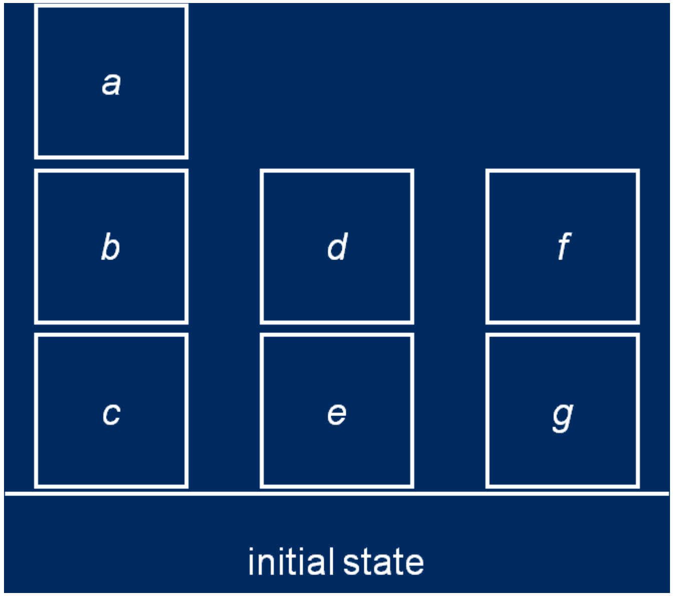
- `block(X).`
- `on(X,Y).`

Defined predicates:

- ```
  tower([X]) :- on(X,table).
  tower([X,Y|T]) :- on(X,Y),tower([Y|T]).
  ```
- `clear(X) :- block(X), not(on(Y,X)).`
  ```

Representing the Initial State

Using the* on(X,Y) *predicate we can represent the initial state.




```
beliefs{
   on(a,b),
   on(b,c),
   on(c,table),
   on(d,e),
   on(e,table),
   on(f,g),
   on(g,table).
}
```

Initial belief base of agent

Representing the Blocks World

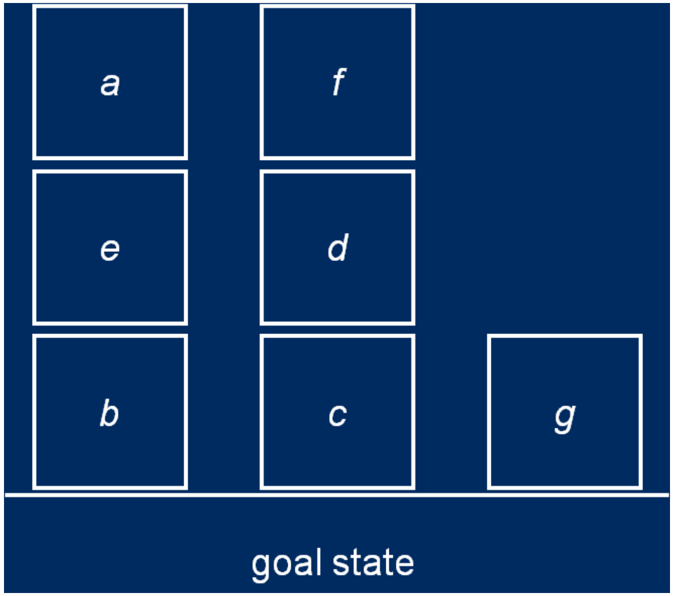
- What about the rules we defined before?
- Insert clauses that do not change into the knowledge base.

```
knowledge{
  block(X) :- on(X,Y).
  clear(X) :- block(X), not(on(Y,X)).
  clear(table).
  tower([X]) :- on(X,table).
  tower([X,Y|T]) :- on(X,Y), tower([Y|T]).
}
```

Static knowledge base of agent

Representing the Goal State

Using the* on(X,Y) *predicate we can represent the goal state.




```
goals{
   on(a,e),
   on(b,table),
   on(c,table),
   on(d,c),
   on(e,b),
   on(f,d),
   on(g,table).
}
```

Initial goal base of agent

One or Many Goals

In the goal base using the comma- or period-separator makes a difference!

```
goals{
   on(a,table).
   on(b,a).
   on(c,b).
}
```

```
goals{
   on(a,table),
   on(b,a),
   on(c,b).
}
```

- Left goal base has three goals, right goal base has single goal.
- Single goal: conjuncts have to be achieved at the same time

Mental State of GOAL Agent

```
knowledge{
   block(X) :- on(X,_).
   clear(X) :- block(X), not(on(Y,X)).
   clear(table).
   tower([X]) :- on(X,table).
   tower([X,Y|T]) :- on(X,Y), tower([Y|T]).
}
beliefs{
   on(a,b), on(b,c), on(c,table), on(d,e), on(e,table),
   on(f,g), on(g,table).
}
goals{
  on(a,e), on(b,table), on(c,table), on(d,c), on(e,b),
   on(f,d), on(g,table).
}
```

Initial mental state of agent

Inspecting the Belief & Goal Base

- Operator `bel(φ)` to inspect the belief base.
- Operator `goal(φ)` to inspect the goal base.
  - Where φ is a Prolog conjunction of literals.
- Examples:
  - `bel(clear(a), not(on(a,c))).`
  - `goal(tower([a,b])).`

Inspecting the Belief Base

- `bel(φ)` succeeds if φ follows from the *belief base in combination with the knowledge base*.

```
knowledge{
  block(X) :- on(X,_).
  clear(X) :- block(X), not(on(Y,X)).
  clear(table).
  tower([X]) :- on(X,table).
  tower([X,Y|T]) :- on(X,Y), tower([Y|T]).
}
beliefs{
  on(a,b), on(b,c), on(c,table), on(d,e), on(e,table),
  on(f,g), on(g,table).
}
```

- Example: `bel(clear(a), not(on(a,c)))` succeeds
- Condition φ is evaluated as a Prolog query.

Inspecting the Goal Base

Use the* goal(...) *operator to inspect the goal base.

- `goal(φ)` succeeds if φ follows from <u>one of the goals in the goal base</u> *in combination with the knowledge base*.

```
knowledge{
  block(X) :- on(X,_).
  clear(X) :- block(X), not(on(Y,X)).
  clear(table).
  tower([X]) :- on(X,table).
  tower([X,Y|T]) :- on(X,Y), tower([Y|T]).
}
goals{
 on(a,e), on(b,table), on(c,table), on(d,c), on(e,b),
  on(f,d), on(g,table).
}
```

- Example: `goal(clear(a))` succeeds. but <u>not</u> `goal(clear(a),clear(c))`.

Why a Separate Knowledge Base?

- Concepts defined in KB can be used in combination with both the belief and goal base.
- *Example*
  - Since agent believes `on(e,table),on(d,e)` infer: agent believes `tower([d,e])`.
  - If agent wants `on(a,table),on(b,a)` infer: agent wants `tower([b,a])`.
- Knowledge base introduced to avoid duplicating clauses in belief and goal base.

Combining Beliefs and Goals

Useful to combine the `bel(…)` and `goal(…)` operators.

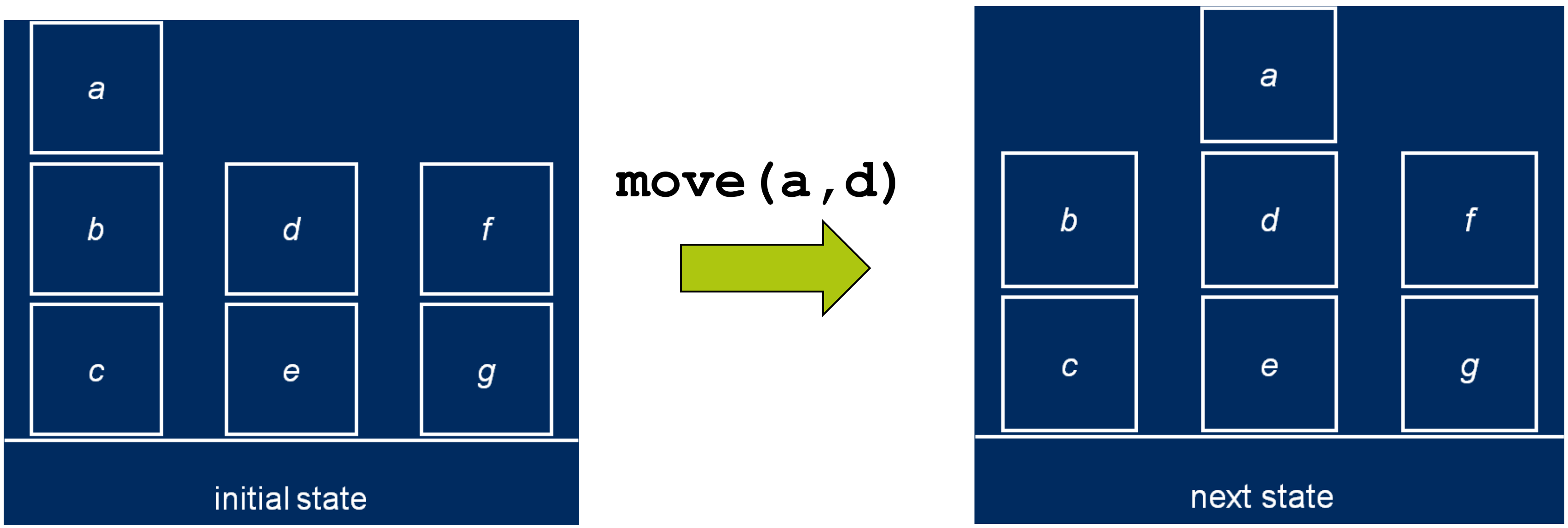
- Achievement goals
  - **`a-goal(φ) = goal(φ), not(bel(φ))`**
    - Agent only has an achievement goal if it does not believe the goal has been reached already.
    - E.g., if belief base is {p.} and goal base is {p,q.}, a-goal(q) but not a-goal(p) holds

- Goal achieved
  - **`goal-a(φ) = goal(φ), bel(φ)`**
    - A (sub)-goal φ has been achieved if the agent believes φ.

8.

Action Specification & Action Selection

Actions Change Environment…



…and Require Updating Mental States: Beliefs

- To ensure adequate beliefs after performing an action the belief base needs to be updated (and possibly the goal base).

```
beliefs{
   on(a,b),
   on(b,c),
   on(c,table),
   on(d,e),
   on(e,table),
   on(f,g),
   on(g,table).
}
```

```
beliefs{
   on(a,d),
   on(b,c),
   on(c,table),
   on(d,e),
   on(e,table),
   on(f,g),
   on(g,table).
}
```

  - *Add effects* to belief base: insert **`on(a,d)`** after **`move(a,d)`**.
  - *Delete* old beliefs: delete **`on(a,b)`** after **`move(a,d)`**.

…and Require Updating Mental States: Goals

- If a goal has been (believed to be) completely achieved, the goal is removed from the goal base.

```
beliefs{
 on(a,table), on(b,table).
}
goals{
 on(a,b), on(b,table).
}
```

move(a,b) →

```
beliefs{
 on(a,b), on(b,table).
}
goals{

}
```

- Default update implements a blind commitment strategy.
- Goal base updates as "side effect" of belief base updates

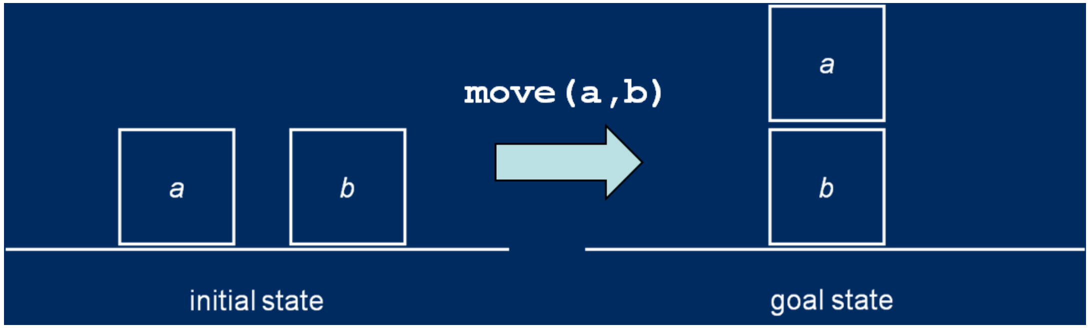
Action Specifications

- Actions in GOAL have preconditions and postconditions (STRIPS-style)
- Executing an action in GOAL means:
  - Preconditions are conditions that need to be true:
    - Check preconditions on the belief base.
  - Postconditions (effects) are add/delete lists:
    - Add positive literals in the postcondition
    - Delete negative literals in the postcondition

```
move(X,Y){
    pre { clear(X), clear(Y), on(X,Z), not( on(X,Y) ) }
    post { not(on(X,Z)), on(X,Y) }
}
```

Actions Specifications

```
move(X,Y){
 pre { clear(X), clear(Y), on(X,Z), not( on(X,Y) )}
 post { not(on(X,Z)), on(X,Y) }
}
```



Example: `move(a,b)`

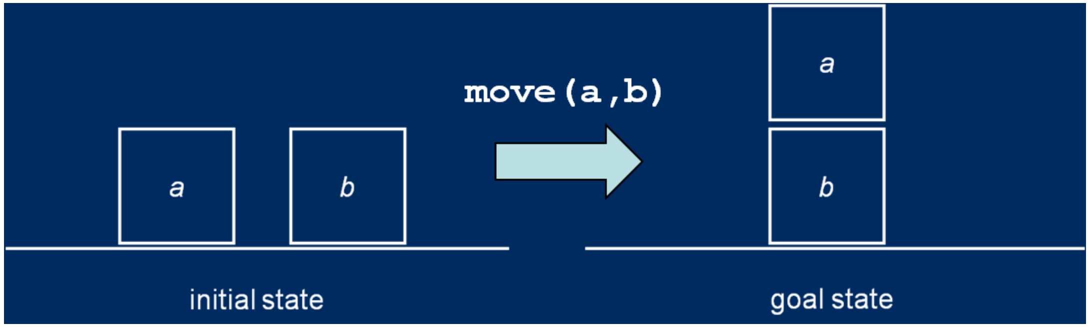
- Check: `clear(a), clear(b), on(a,Z), not(on(a,b))`
- Remove: `on(a,Z)`
- Add: `on(a,b)`

`Z = table`

Actions Specifications

```
move(X,Y){
  pre { clear(X), clear(Y), on(X,Z) }
  post { not(on(X,Z)), on(X,Y) }
}
```

Example: `move(a,b)`




```
beliefs{
  on(a,table),
  on(b,table).
}
```

```
beliefs{
  on(b,table).
  on(a,b).
}
```

Built-in Actions

Adopting and dropping goals:

- adopt(<conjunction of positive literals>)
  meaning: add a **new** goal to goal base (if not already **implied** by a goal)
- drop(<conjunction>)
  meaning: remove **all** goals that imply <conjunction> from the goal base

Inserting and deleting beliefs:

- insert(<conjunction>)
- delete(<conjunction>)

Drop Action

`drop(on(b,a), not(on(c,table)))`

```
knowledge{
 block(X) :- on(X, Y).
 clear(X) :- block(X), not(on(Y,X)).
 clear(table).
 tower([X]) :- on(X, table).
 tower([X,Y|T]):- on(X,Y),tower([Y|T]).
}
goals{
 on(a,table), on(b,a), on(c,b),
 on(d,table), on(e,table), on(f,e),
 on(g,f), on(h,g), on(i,h).
}
```

- Is goal in goal base dropped?
- Check: does goal imply `on(b,a), not(on(c,table))` ?
- A: Yes, so goal is removed by drop action.

Action Selection in Agent-Oriented Programming

- How do humans choose and/or explain actions?
- Examples:
  - I believe it rains; so, I will take an umbrella with me.
  - I go to the video store because I want to rent I-robot.
  - I don't believe busses run today so I take the train.
- BDI not only for explaining & predicting, but also for programming!
- Use intuitive common sense concepts:

beliefs + goals => action

Selecting Actions: Action Rules

- Action rules are used to define a strategy for action selection.
- Defining a strategy for blocks world:
  - If constructive move can be made, make it.
  - If block is misplaced, move it to table.

```
program{
    if bel(tower([Y|T])), a-goal(tower([X,Y|T])) then move(X,Y).
    if a-goal(tower([X|T])) then move(X,table).
}
```

- What happens:
  - Check condition, e.g. can **`a-goal(tower([X|T]))`** be derived given current mental state of agent?
  - Yes, then (potentially) select **`move(X,table)`**.

9.

Sensing & Environments

Sensing

- Agents need sensors to:
  - explore the environment when they have incomplete information (e.g. Wumpus World)
  - keep track of changes in the environment that are not caused by itself
- GOAL agents sense the environment through a perceptual interface defined between the agent and the environment
  - Environment generates percepts
  - Environment Interface Standard: EIS (Hindriks et al.)

Percept Base

- Percepts are received by an agent in its percept base.

- The reserved keyword `percept` is wrapped around the percept content, e.g. `percept(block(a))`.

- Not automatically inserted into beliefs!

Processing Percepts

- The percept base is refreshed, i.e. emptied, every reasoning cycle of the agent.
- Agent has to decide what to do when it perceives something, i.e. receives a percept.
- Use percepts to update agent's mental state:
  - Ignore the percept
  - Update the beliefs of the agent
  - Adopt/drop a new goal

Updating Agent's Mental State

One way to update beliefs with percepts:

- First, delete everything agent believes.
  Example: remove all `block` and `on` facts.
- Second, insert new information about current state provided from percepts into belief base.
  Example: insert `block` and `on` facts for every `percept(block(…))` and `percept(on(…))`.

Assumes that environment is **fully observable** with respect to `block` and `on` facts.
Downside: not very efficient…

Percept Update Pattern

A typical pattern for updating is:

Rule 1

If the agent

- **perceives** block X is on top of block Y, and
- does **not believe** that X is on top of Y

Then **insert** `on(X,Y)` into the belief base.

Rule 2

If the agent

- **believes** that X is on top of Y, and
- does **not perceive** block X is on top of block Y

Then **remove** `on(X,Y)` from the belief base.

Percepts and Event Module

- Percepts are processed in GOAL by means of event rules, i.e. rules in the event module.

```
event module{
 program{
    <…
      rules
    …>
 }
}
```

- Event module is executed every time that agent receives new percepts.

Implementing Pattern Rule 1

INCORRECT!

Rule 1

If the agent

- **perceives** block X is on top of block Y, and
- does **not believe** that X is on top of Y

Then **insert** `on(X,Y)` into the belief base.

```
event module {
 program{
   % assumes full observability.
   if bel(percept(on(X,Y)), not(on(X,Y))) then insert(on(X,Y)).
   …
 }
}
```

Note: percept base is inspected using the bel operator, e.g. `bel(percept(on(X,Y)))`.

Implementing Pattern Rule 1

Rule 1

If the agent **perceives** block X is on top of block Y, and does **not believe** that X is on top of Y, then **insert** `on(X,Y)` into the belief base.

*We want to apply this rule **for all** percept instances that match it!*

Content Percept Base

```
percept(on(a,table))
percept(on(b,table))
percept(on(c,table))
percept(on(d,table))
…
```

```
event module {
 program{
   % assumes full observability.
   forall bel(percept(on(X,Y)), not(on(X,Y))) do insert(on(X,Y)).
   …
 }
}
```

Implementing Pattern Rule 2

Rule 2

If the agent

- **believes** that X is on top of Y, and
- does **not perceive** block X is on top of block Y

Then **remove** `on(X,Y)` from the belief base.

```
event module {
 program{
   % assumes full observability.
   forall bel(percept(on(X,Y)), not(on(X,Y))) do insert(on(X,Y)).
   forall bel(on(X,Y), not(percept(on(X,Y)))) do delete(on(X,Y)).
 }
}
```

1. We want that **all** rules are applied!
   By default the event module applies all rules in linear order.
2. Note that none of these rules fires if nothing changed.

Initially… Agent Has No Beliefs

- In most environments an agent initially has no information about the state of the environment, e.g. Tower World, Wumpus World, …
- Represented by an empty belief base:

```
beliefs{
}
```

- There is no need to include a belief base in this case in a GOAL agent.
- It is ok to simply have no belief base section.

Summarizing

- Two types of rules:
  - `if <cond> then <action>.`
    is applied at most once (if multiple instances chooses randomly)
  - `forall <cond> do <action>.`
    is applied once *for each* instantiation of parameters that satisfy condition.
- Main module by default:
  - checks rules in **linear order**
  - applies **first** applicable rule (also checks action precondition!)
- Event module by default:
  - Checks rules in **linear order**
  - Applies **all** applicable rules (rules may enable/disable each other!)
- Program section modifiers: `[order=random]`, `[order=linear]`, `[order=linearall]`, `[order=randomall]`
- Built-in actions: insert, delete, adopt, drop.

10.

Instantaneous & Durative Actions

Instantaneous versus Durative

- Instantaneous actions
  Actions in the *Blocks World* environment are instantaneous, i.e. they do not take time.
  Wumpus World actions are of this type as well.

- Durative actions
  Actions in the *Tower World* environment take time.
  When a GOAL agent sends an action to such an environment, the action will not be completed immediately.

Durative Actions and Sensing

- While durative actions are performed an agent may receive percepts.
- Useful to monitor progress of action.
- UT2004 Example:

  Other bot is perceived while moving.

Specifying Durative Actions

- delayed effect problem
- solution: “no” postcondition
  - results of action are handled by event rules
- Postcondition may be “empty”: `post { }`
- Better practice is to indicate that you have not forgotten to specify it by using `post { true }`.

11.

GOAL: Program Structure

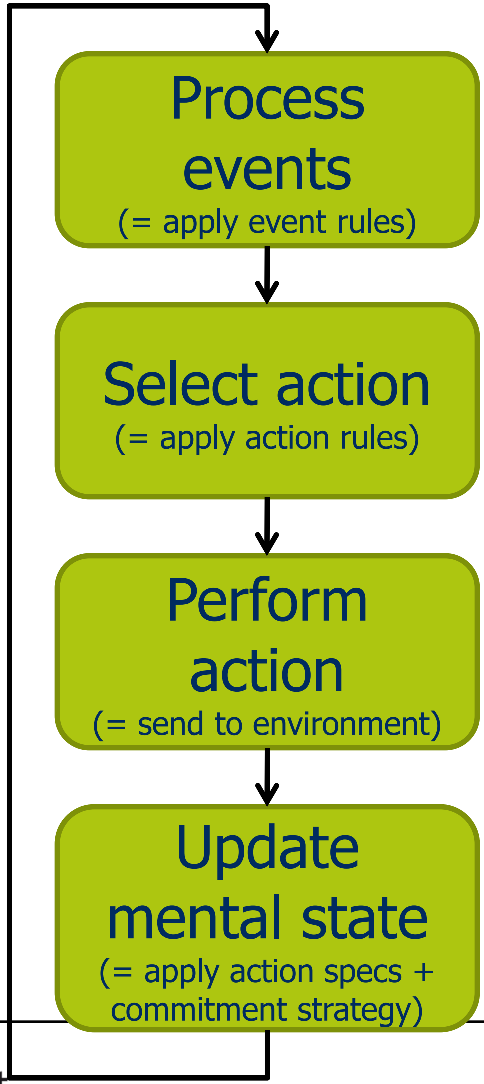
Structure of GOAL Program

```
init module {
  <initialization of agent>
}

main module{
  <action selection strategy>
}

event module {
 <percept processing>
}
```

GOAL Interpreter Cycle



Also called reasoning or deliberation cycle.

GOAL's cycle is a classic sense-plan-act cycle.

Sections in Modules

1. knowledge{...}
2. beliefs{...}
3. goals{...}
4. program{...}
5. actionspec{...}

- Init module: all sections optional, globally available
- Main & event module: 2 not allowed; 4 obligatory; 1,3,5: optional

- At least event or main module should be present

12.

Modularity & Multiple Agents

Modularity in Agent Programming

- Central issue in software engineering
- Increased understandability of programs
- Busetta et al. (ATAL'99): capability
  - cluster of components of a cognitive agent
- Braubach et al. (ProMAS'05): extension of capability notion
- Van Riemsdijk et al. (AAMAS'06): goal-oriented modularity
  - idea: modules encapsulate information on how to achieve a goal
  - dispatch (sub)goal to module

Modules in GOAL

- User-defined modules, next to init, main and event
- Idea: focus attention on (part of) goal
- Use action rules to call module
  - if goal condition in action rules, corresponding goals become (local) goals of module
  - different exit policies: after doing one action; when local goals have been achieved; when no actions can be executed anymore; using explicit exit-module action
- See also Hindriks (ProMAS'07)

Multi-Agent System in GOAL

- .mas2g file: launch rules to start multiple agents
- action `send(Receiver, Content)` to send messages
  - mailbox semantics: inspected using `bel` operator
- declarative, imperative and interrogative "moods"
- Hindriks, Van Riemsdijk (ProMAS'09): communication semantics based on mental models

Not Discussed

- …