Overview Other APLs & Architectural Considerations

Overview

- Other APLs
  - 2APL
  - Jason (short)
  - ConGolog (short), IndiGolog
  - Jadex (short)
  - JACK (short)
  - CLAIM (short)
- Some Architectural Considerations
- Research Themes
- References

Agent Programming Languages: An Overview

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.)

2APL: A Practical Agent Programming Language

Features of 2APL (1)

- Programming Constructs
  - Multi-Agent System Which and how many agents to create? Which environments? Which agent can access which environment?
  - Individual Agent Beliefs, Goals, Plans, Events, Messages
- Programming Principles and Techniques
  - Abstraction Procedures and Recursion in Plans
  - Error Handling Plan Failure and their revision by Internal Events, Execution of Critical Region of Plans
  - Legacy Systems Environment and External Actions
  - Encapsulation Including 2APL files in other 2APL files
  - Autonomy Adjustable Deliberation Process

Features of 2APL (2)

- Integrated Development Environment
  - 2APL platform is Built on JADE and uses related tools
  - Editor with High-Lighting Syntax
  - Monitoring mental attitudes of individual agents, their reasoning and communications
  - Executing in one step or continuous mode
  - Visual Programming of the Deliberation Process

2APL Syntax: Programming Multi-Agent System

Agents, Numbers, and Access to Environment

$\texttt{agentname}_1$: $\texttt{filename}_1\texttt{.2apl N @env}_1\texttt{,...,env}_k$

$\vdots$

$\texttt{agentname}_k$: $\texttt{filename}_k\texttt{.2apl M @env'}_1\texttt{,...,env'}_l$

- $\texttt{agentname}_i$ is the name of the agent to be created
- `filname.2apl` is the name of the 2APL file that is used to initialise agent
- `N` is the number of agents to be created based on one 2APL file `filename.2apl`. When $N > 1$, the name of the created agents are indexed with a unique number
- $\texttt{env}_1\texttt{,...,env}_k$ are the names of the environments

2APL Syntax: Programming Multi-Agent System

Example (Block World)

```
explorer   :   explorerSpec.2apl   2   @bw , @db
carrier    :   carrierSpec.2apl    4   @bw
```

The explorer agents $\texttt{explorer}_1$ and $\texttt{explorer}_2$ find the objects by either searching the blockworld `bw` or querying the database `db` (where the information about objects are stored).

The carrier agents $\texttt{carrier}_1, \ldots, \texttt{carrier}_4$ receive the information about object locations from explorer agents and carry them to a depot position.

2APL Syntax: Programming Individual Agents (1)

General Scheme

$$
\begin{array}{lcl}
\langle \textit{Program} \rangle & ::= & \{\text{"Include:"}\ \langle \textit{ident} \rangle \\
 & | & \text{"Beliefupdates:"}\ \langle \textit{BelUpSpec} \rangle \\
 & | & \text{"Beliefs:"}\ \langle \textit{beliefs} \rangle \\
 & | & \text{"Goals:"}\ \langle \textit{goals} \rangle \\
 & | & \text{"Plans:"}\ \langle \textit{plans} \rangle \\
 & | & \text{"PG-rules:"}\ \langle \textit{pgrules} \rangle \\
 & | & \text{"PC-rules:"}\ \langle \textit{pcrules} \rangle \\
 & | & \text{"PR-rules:"}\ \langle \textit{prrules} \rangle\ \}
\end{array}
$$

2APL Syntax: Programming Individual Agents

Example (Cleaning Block World and Collecting Gold)

```
Beliefupdates:
{ dirt(X,Y) } PickUpDirt() { not dirt(X,Y) }
{ pos(X,Y) } goto(V,W) {not pos(X,Y), pos(V,W)}
Beliefs:     post(5,5).
             dirt(3,6).
             clean(world) :- not dirt(X,Y).
Goals:       hasGold(2) and clean(world) ,
             hasGold(5)
PG-rules:    clean(world) <- dirt(X,Y) |
             { goto(X,Y);PickUpDirt() }
PC-rules:    event(goldAt(X,Y)) <- true |
             { goto(X,Y); PickUpGold() }
```

2APL Syntax: Programming Individual Agents (2)

Mental Attitudes: Updates, Beliefs, Goals and Plans

$$\langle \mathit{BelUpSpec} \rangle \quad ::= \quad (\ \text{"\{"} \langle \mathit{belquery} \rangle\ \text{"\}"}\ \langle \mathit{belUp} \rangle \text{"\{"} \langle \mathit{literals} \rangle \text{"\}"}\ )+$$

$$\begin{array}{lll} \langle \mathit{beliefs} \rangle & ::= & (\ \langle \mathit{ground\_atom} \rangle\ \text{"."} \\ & | & \langle \mathit{atom} \rangle\ \text{": -"}\ \langle \mathit{literals} \rangle \text{"."}\ )+ \end{array}$$

$$\begin{array}{lll} \langle \mathit{goals} \rangle & ::= & \langle \mathit{goal} \rangle\ \{\text{","}\ \langle \mathit{goal} \rangle\ \} \\ \langle \mathit{goal} \rangle & ::= & \langle \mathit{ground\_atom} \rangle\ \{\ \text{"and"}\ \langle \mathit{ground\_atom} \rangle\ \} \end{array}$$

$$\langle \mathit{plans} \rangle \quad ::= \quad \langle \mathit{plan} \rangle\ \{\ \text{","}\ \langle \mathit{plan} \rangle\ \}$$

2APL Syntax: Programming Individual Agents (3)

Plans and Actions

$$
\begin{array}{lcl}
\langle \mathit{plan} \rangle & ::= & \text{"skip"} \mid \langle \mathit{belUp} \rangle \mid \langle \mathit{test} \rangle \\
 & \mid & \langle \mathit{abstractaction} \rangle \\
 & \mid & \langle \mathit{adoptgoal} \rangle \mid \langle \mathit{dropgoal} \rangle \\
 & \mid & \langle \mathit{externalaction} \rangle \mid \langle \mathit{sendaction} \rangle \\
 & \mid & \langle \mathit{whileplan} \rangle \mid \langle \mathit{ifplan} \rangle \\
 & \mid & \langle \mathit{sequenceplan} \rangle \mid \langle \mathit{atomicplan} \rangle \\
 & & \\
\langle \mathit{test} \rangle & ::= & \text{"B("} \langle \mathit{belquery} \rangle \text{")"} \mid \text{"G("} \langle \mathit{goalquery} \rangle \text{")"} \\
 & \mid & \langle \mathit{test} \rangle \;\&\; \langle \mathit{test} \rangle \\
 & & \\
\langle \mathit{externalaction} \rangle & ::= & \text{"@"} \langle \mathit{ident} \rangle \text{"("} \langle \mathit{atom} \rangle \text{","} \langle \mathit{Var} \rangle \text{")"} \\
 & & \\
\langle \mathit{sendaction} \rangle & ::= & \text{"Send("} \langle \mathit{iv} \rangle \text{","} \langle \mathit{iv} \rangle \text{","} \langle \mathit{atom} \rangle \text{")"}
\end{array}
$$

2APL Syntax: Programming Individual Agents (4)

Composite Plans

$\langle$*whileplan*$\rangle$	::=	"while" $\langle$*test*$\rangle$ "do" $\langle$*scopeplan*$\rangle$
$\langle$*ifplan*$\rangle$	::=	"if" $\langle$*test*$\rangle$ "then" $\langle$*scopeplan*$\rangle$ ["else" $\langle$*scopeplan*$\rangle$]
$\langle$*sequenceplan*$\rangle$	::=	$\langle$*plan*$\rangle$ ";" $\langle$*plan*$\rangle$
$\langle$*scopeplan*$\rangle$	::=	"{" $\langle$*plan*$\rangle$ "}"
$\langle$*atomicplan*$\rangle$	::=	"[" $\langle$*plan*$\rangle$ "]"

2APL Syntax: Programming Individual Agents (5)

Reasoning Rules

$$
\begin{array}{lcl}
\langle \textit{pgrules} \rangle & ::= & \langle \textit{pgrule} \rangle + \\
\langle \textit{pgrule} \rangle & ::= & [\langle \textit{goalquery} \rangle] \text{"}< -\text{"} \langle \textit{belquery} \rangle \text{"}|\text{"} \langle \textit{plan} \rangle \\
\\
\langle \textit{pcrules} \rangle & ::= & \langle \textit{pcrule} \rangle + \\
\langle \textit{pcrule} \rangle & ::= & \langle \textit{atom} \rangle \text{"}< -\text{"} \langle \textit{belquery} \rangle \text{"}|\text{"} \langle \textit{plan} \rangle \\
\\
\langle \textit{prrules} \rangle & ::= & \langle \textit{prrule} \rangle + \\
\langle \textit{prrule} \rangle & ::= & \langle \textit{planvar} \rangle \text{"}< -\text{"} \langle \textit{belquery} \rangle \text{"}|\text{"} \langle \textit{planvar} \rangle
\end{array}
$$

Part II

AgentSpeak(L) & Jason

AgentSpeak(L)

- Originally proposed by Rao [MAAMAW 1996] as an (elegant) abstract agent programming language
- Programming language for BDI agents (reactive planning systems)
- Based on PRS and the work on BDI logics
- Various extentions were necessary to make it more practical
- ***Jason*** implements the operational semantics of an extended version of AgentSpeak
- ***Jason* is jointly developed with Jomi F. Hübner (FURB, Brazil)**

Scenario for a Running Example

- Abstract version of a Mars exploration scenario: a typical day of activity of an autonomous Mars rover
- Typical instructions sent to the rover by the ground team:
  1. Back up to the rock named Soufflé
  2. Place the arm with the spectrometer on the rock
  3. Do extensive measurements on the rock surface
  4. Perform a long traverse to another rock
- It turned out that the robot was not correctly positioned, so scientific data was lost
- Green patches on rocks indicate good science opportunity
- Batteries only work while there is sunlight (“sol” is a Martian day)
- Detailed program used in the experiments had 25 plans

Examples of Plans

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

+!traverse(Coords) :
    safe_path(Coords) <-
        move_towards(Coords).

+!traverse(Coords) :
    not safe_path(Coords) <-
        ...
```

Examples of Plans (II)

```
+!examine(Rock) :
    correctly_positioned(Rock) <-
        place_spectrometer(Rock);
        !extensive_measurements(Rock).

+!examine(Rock) :
    not correctly_positioned(Rock) <-
        !correctly_positioned(Rock);
        !examine(Rock).
```

Language Extensions (I)

- Annotated predicate:

$$\mathtt{ps}(\mathtt{t}_1,\ldots,\mathtt{t}_\mathtt{n})[\mathtt{a}_1,\ldots,\mathtt{a}_\mathtt{m}]$$

  where a_i are first-order terms (these have no annotations)
- in the belief base, all predicates have a special annotation

$$\mathtt{source}(\mathtt{s}_\mathtt{i})$$

  where $s_i \in \{\mathtt{self}, \mathtt{percept}, id\}$, and *id* is any agent label (i.e., name)

Example (belief annotations)

```
blue(box1)[source(ag1)].
red(box1)[source(percept)].
colourblind(ag1)[source(self),degOfCert(0.7)].
lier(ag1)[source(self),degOfCert(0.2)].
```

Language Extensions (II)

- Plan labels also can have annotations
- Easy to write (in Java) selection functions that use information about the plans contained in such annotations
- Annotation can also be dynamically changed in instances of plans (intentions)
  - this can be used, e.g., to update the priority that needs to be given to a certain plan

Example (plan with annotated label)

```
anotherLabel[chanceOfSuccess(0.7),
    usualPayoff(0.9), anyOtherProperty] ->
+b(X) : c(t) <- a(X).
```

Language Extensions (III)

- Strong negation (operator ~)

Example (strong negation)

```
+!leave(home)
    :  not raining & not ~raining
       <- open(curtains); ...

+!leave(home)
    :  not raining & not ~raining
       <- .send(mum,askIf,raining); ...
```

- Deletion events used for handling plan failures

Example (an agent blindly committed to g)

```
+!g :  g <- true.

+!g :  ...  <- ...  !g.

-!g :  true <- !g.
```

Language Extensions (IV)

- Internal actions can be defined by the user in Java (or other programming languages)

  `libName.actionName(...)`

- Standard (pre-defined) internal actions have an empty library name
- Internal action for communication: `.send(`*r*, *ilf*, *pc*`)` where *ilf* $\in$ {`tell`, `untell`, `achieve`, `unachieve`, `tellHow`, `untellHow`, `askIf`, `askOne`, `askAll`, `askHow`}
- Some other standard internal actions:
  - `.desire(`*literal*`)`
  - `.intend(`*literal*`)`
  - `.dropDesires(`*literal*`)`
  - `.dropIntentions(`*literal*`)`
  - print, sort, list operations, etc.

MAS Configuration File

- ***Jason*** has a simple language for defining a multi-agent system, where each agent runs it's own AgentSpeak interpreter, and an environment can be given by a Java class

```
MAS Auction {
    infrastructure: Saci
    environment: AuctionEnv
    agents: ag1; ag2; ag3;
}
```

MAS Configuration (II)

- System *Architecture* options: Centralised or Saci
- Easy to specify in which *host* agents and the environment will run

```
agents:
    ag1 at host1.dur.ac.uk;
```

- Explicitly specifying the file where the agent's *source code* is to be found

```
agents: ag1 file1;
```

- Indicating the *number of instances* of an agent (using the same initial beliefs and plan library)

```
agents: ag1 #10;
```

Customising the Infrastructure

- Users can define a specific way the agent interacts with the multi-agent systems infrastructure
- This is used to customise the way the agent does perception of the environment, receives communication massages, and acts in the environment

In the configuration file:

```
agents: ag1 agentArchClass MyAgArch;
```

Example of customised architecture class:

```
import jason.architecture.*;
public class MyAgArch extends AgentArchitecture {
   public void perceive() {
      System.out.println("Getting percepts!");
      super.perceive();
}  }
```

Customising an Agent Class

- This is used to customise the *selection functions* of the AgentSpeak interpreter and other agent-specific functions
  - Selection functions
  - Belief update and revision
  - Functions defining trust/power relations for processing communication messages
  - Message and action-feedback (from environment) processing priorities

Environments

- In actual deployment, there will normally be a real-world environment where the MAS will be situated
- The AgentArchitecture needs to be customised to get perceptions and act on such environment
- We often want a simulated environment (e.g., to test the MAS)
- This can be done in Java by extending ***Jason***'s `Environment` class and using methods such as `addPercept(String Agent, Literal Percept)`

Part III

(Con)Golog

(Con)Golog

- Created by Levesque, Reiter, Lesperance, ...
- Based on **situation calculus**, *a predicate calculus dialect for representing dynamically changing worlds*
- $do(agt, act, s)$: state resulting from agent agt's performance of action act in state (situation) s
- Formal semantics of actions based on situation calculus, e.g.:
  - primitive actions

$$Do(by(agt, act), s, s') =_{def} \exists s^*(s < do(agt, act, s^*) \leq s')$$

  - test actions

$$Do(\phi?, s, s') =_{def} \exists s^*(s < s^* \leq s' \ \wedge \ \phi(s^*))$$

(Con)Golog

- sequence of actions

$$Do(\delta_1 ; \delta_2, s, s') =_{def} \exists s^* (Do(\delta_1, s, s^*) \wedge Do(\delta_2, s^*, s'))$$

- concurrent actions

$$Do(\delta_1 ` \delta_2, s, s') =_{def} (Do(\delta_1, s, s') \wedge Do(\delta_2, s, s'))$$

- nondeterministic choice of actions

$$Do(\delta_1 | \delta_2, s, s') =_{def} (Do(\delta_1, s, s') \vee Do(\delta_2, s, s'))$$

(Con)Golog

- ConGolog programs are evaluated with a theorem prover
- user provides: (AXIOMS=)
  - precondition axioms (one per action)
  - successor state axioms (one per fluent)
  - specification of the initial state of the world +
  - ConGolog program specifying the behaviour of the agents in the system

(Con)Golog

- execution of the program:
  - prove (constructively)

    $$AXIOMS \models \exists s Do(program, S_0, s)$$

    constructive proof yields binding for variable s:

    $$s = do(agt_n, act_n, \ldots, do(agt_1, act_1, S_0) \ldots)$$

  - send sequence $(agt_1, act_1), \ldots, (agt_n, act_n)$ to the primitive action execution module
- N.B. nondeterministic actions allowed ("sketchy planning")

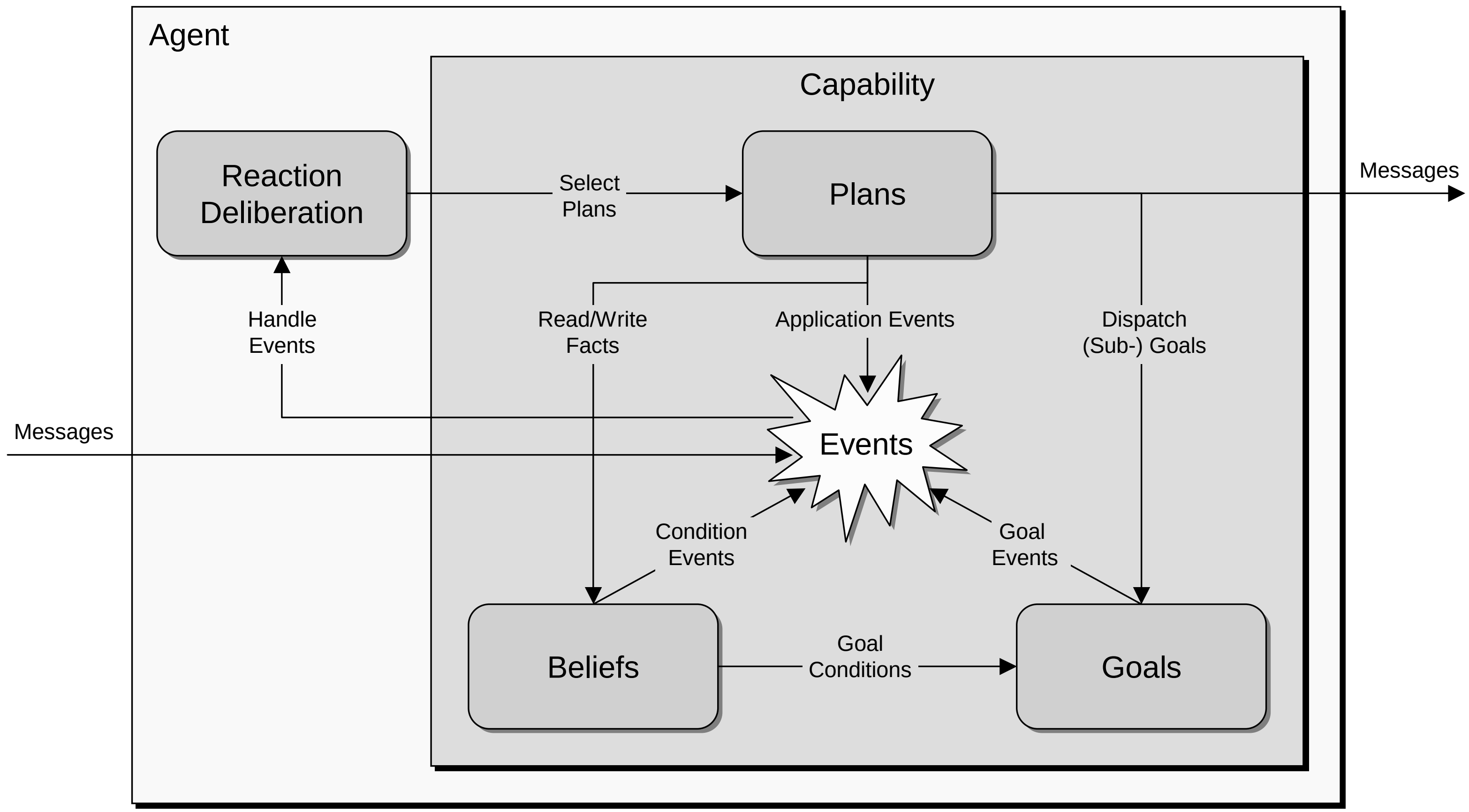
Part IV

Jadex

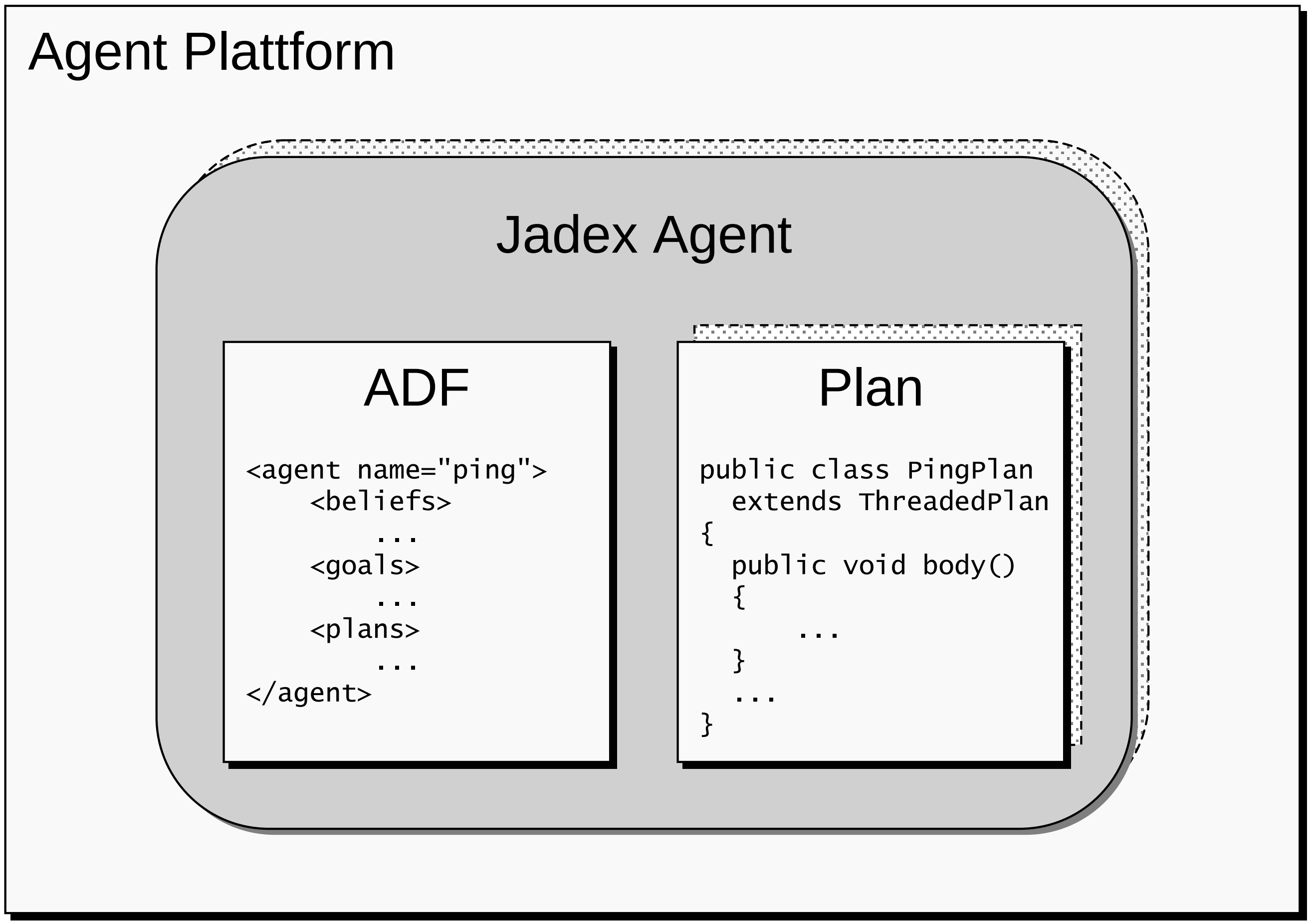
Jadex: Background and Motivation

- Developed by Lars Braubach, Alexander Pokahr, and Winfried Lamersdorf
- Jadex is built on top of the **JADE Platform**
- Jadex is based on the **BDI model**
- Integrate agent theories with **object-orientation** and **XML** descriptions
- Object-oriented representation of BDI concepts
- Explicit representation of **goals** allows reasoning about (manipulation of) goals

Jadex Agent Architecture



Jadex Implementation Model



Jadex: Beliefs

- Central place for knowledge: **accessible to all plans**
- Allows **queries** over the agent's beliefs
- Allows **monitoring** of beliefs and conditions
- No support for **logical reasoning**

Jadex: Goals

- Generic **goal types**
  - **perform** (some action)
  - **achieve** (a specified world state)
  - **query** (some information)
  - **maintain** (reestablish a specified world state whenever violated)
- Are **strongly typed** with
  - name, type, parameters
  - **BDI-flags** enable non-default goal-processing
- Goal **creation/deletion** possibilities
  - initial goals for agents
  - goal creation/drop conditions for all goal kinds
  - top-level / subgoals from within plans

Jadex: Plans

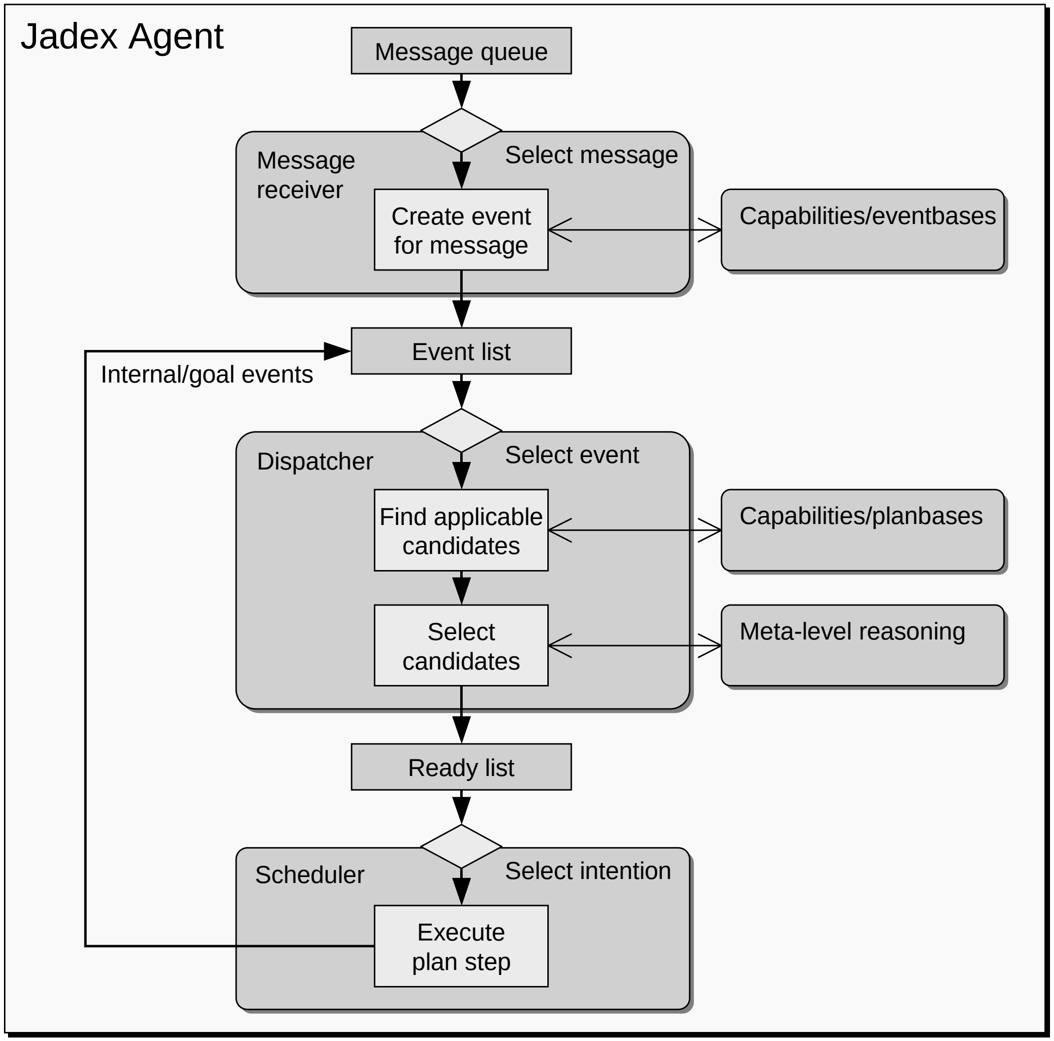
- Represent procedural knowledge
  - Means for goal **achievement** and **reacting to events**
  - Agent has library of pre-defined plans
  - Interleaved stepwise execution
- Realisation of a plan
  - Plan head specified in **ADF (Agent Definition File)**
  - **Plan body** coded in **pure Java**
- Assigning plans to goals/events
  - Plan head indicates ability to handle goals/events
  - Plan context / precondition further refines set of applicable plans

Jadex: Events

Three types of Events:

- **Message event** denotes arrival/sending messages
- **Goal event** denotes a new goal to be processed or the state of an existing goal is changed
- Internal event
  - **Timeout event** denotes a timeout has occurred, e.g., waiting for arrival of messages/achieving goals/waitFor(duration) actions.
  - Execute **plan event** denotes plan to be executed without meta-level reasoning, e.g., plans with triggering condition
  - **Condition-triggered event** is generated when a state change occur that satisfies the trigger of a condition

Jadex Interpreter



Part V

Jack

JACK Agent Language

Extends Java with …

Class Constructs

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

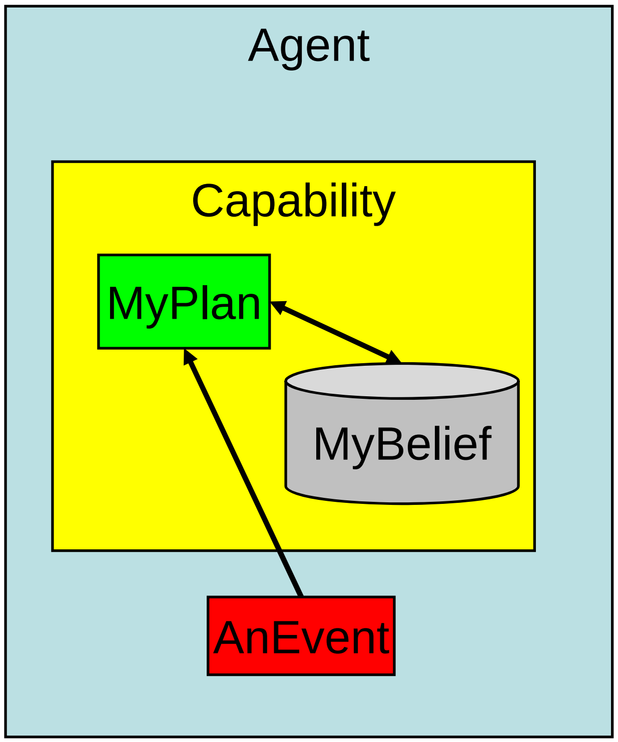
Declarations

- #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

Part VI

CLAIM

CLAIM : a declarative language

- Developed by El Fallah Seghrouchni and Suna
- Cognitive elements
  - Goals, knowledge, capabilities
  - Reasoning : reactivity and pro-activity
- Interaction et mobility
  - Communication primitives
  - **Mobility primitives** (Ambient Calculus)
- Operational semantics
  - Mobility's and interaction's management
  - Appropriate for intelligent agents

Defining agents and classes

```
defineAgent agentName {
        authority = agentName ;
        parent = null | agentName ;
        knowledge = null |  (knowledge;)* ;
        goals = null |  (goal ;)*  ;
        messages = null |  (queueMessage ;)*  ;
        capabilities = null |  (capability ;)*  ;
        processes = null |  (process | )*  ;
        agents = null |  (agentName ;)*  ;
}
```

Intelligent elements

```
knowledge =
agentName(CapabilityName,message,effect)
                    | agentName:class
                    | proposition
```

Example: Prod1:CoffeeProducer;
Producer(Prod1,oro,7);
Producer(Prod2,oro,6);

```
goal = proposition
```

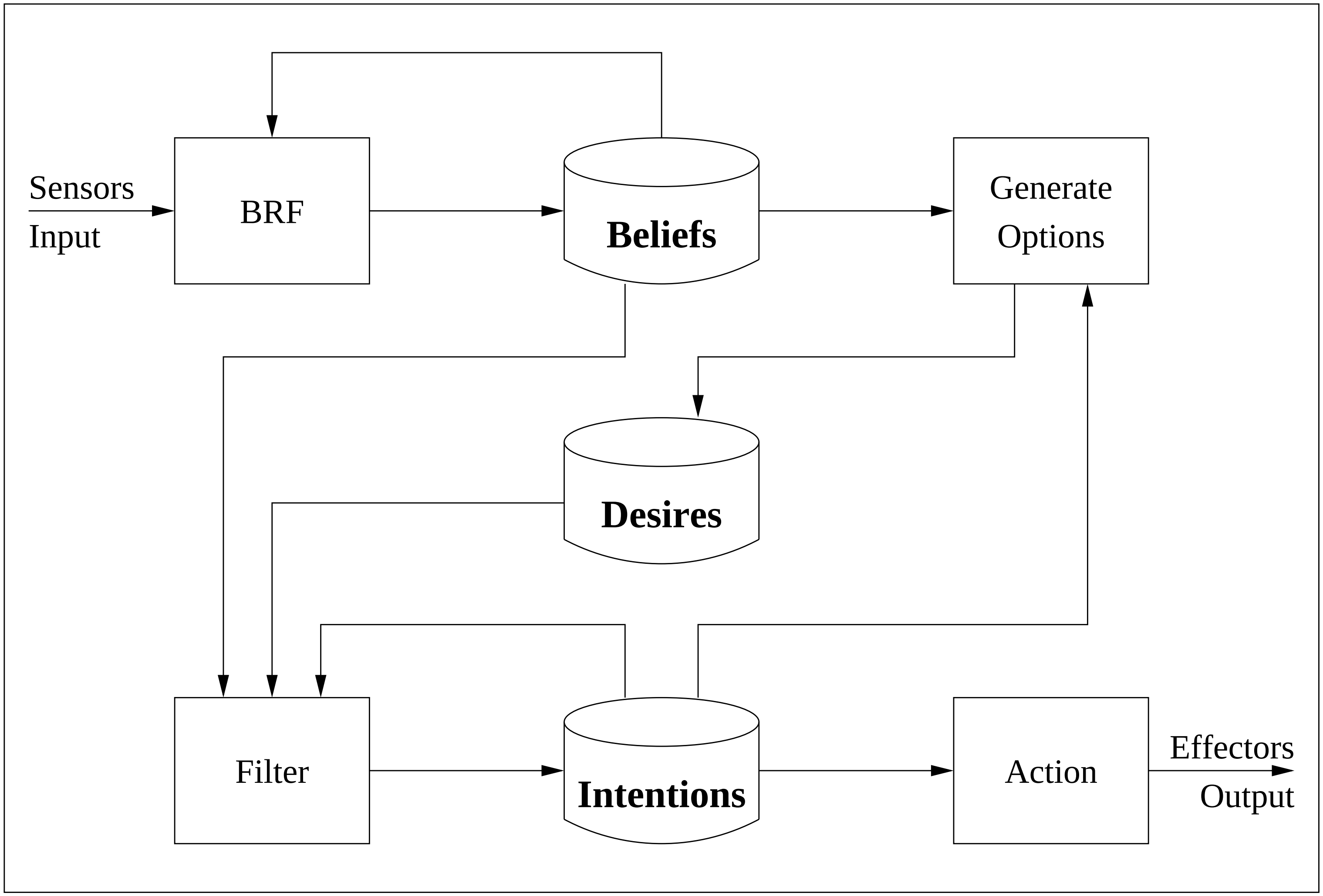
Example: For a Buyer
haveCoffee(oro,7,500)

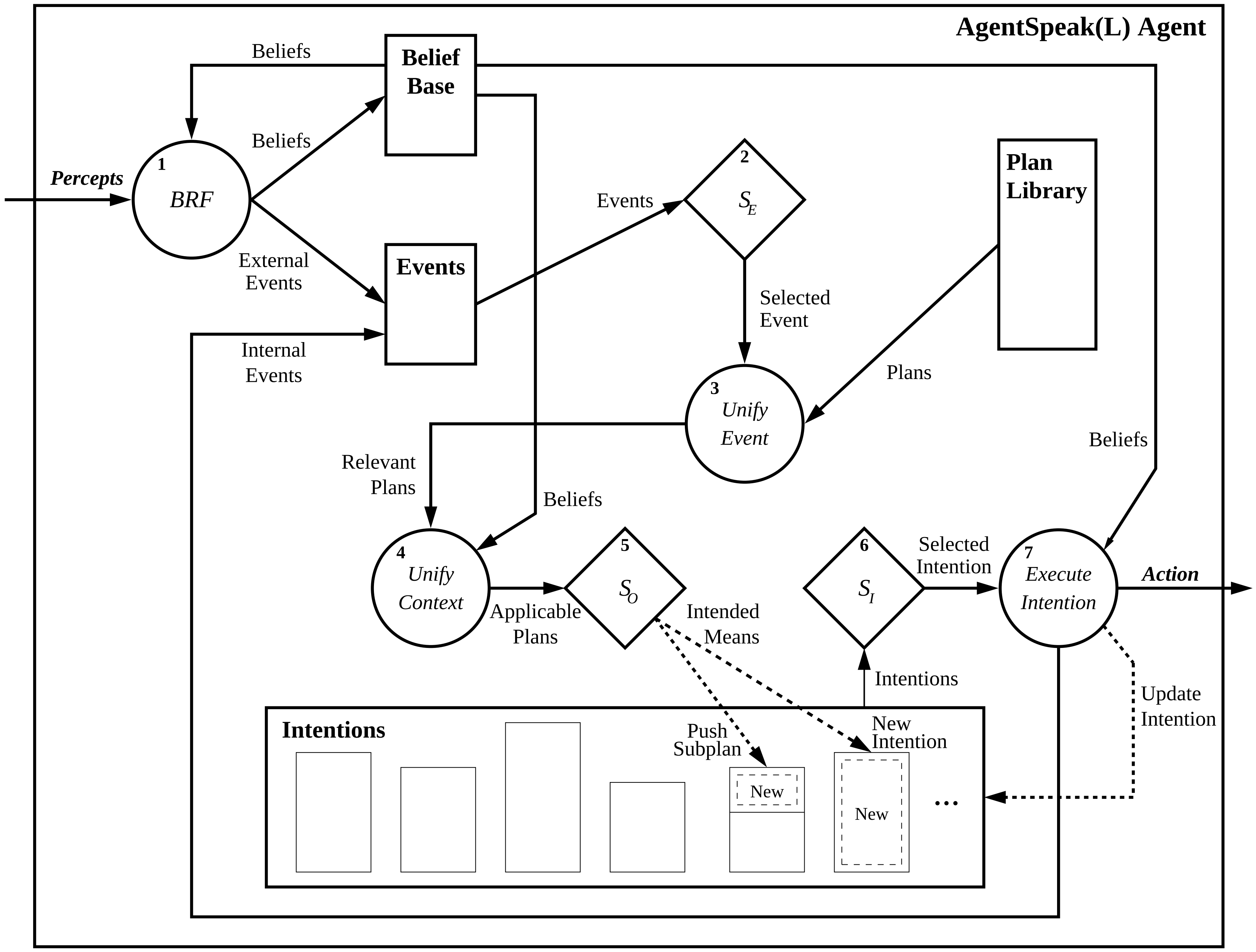
```
process = instructions
```

Example: mobility instruction
moveTo(mobilityArg,agentName)

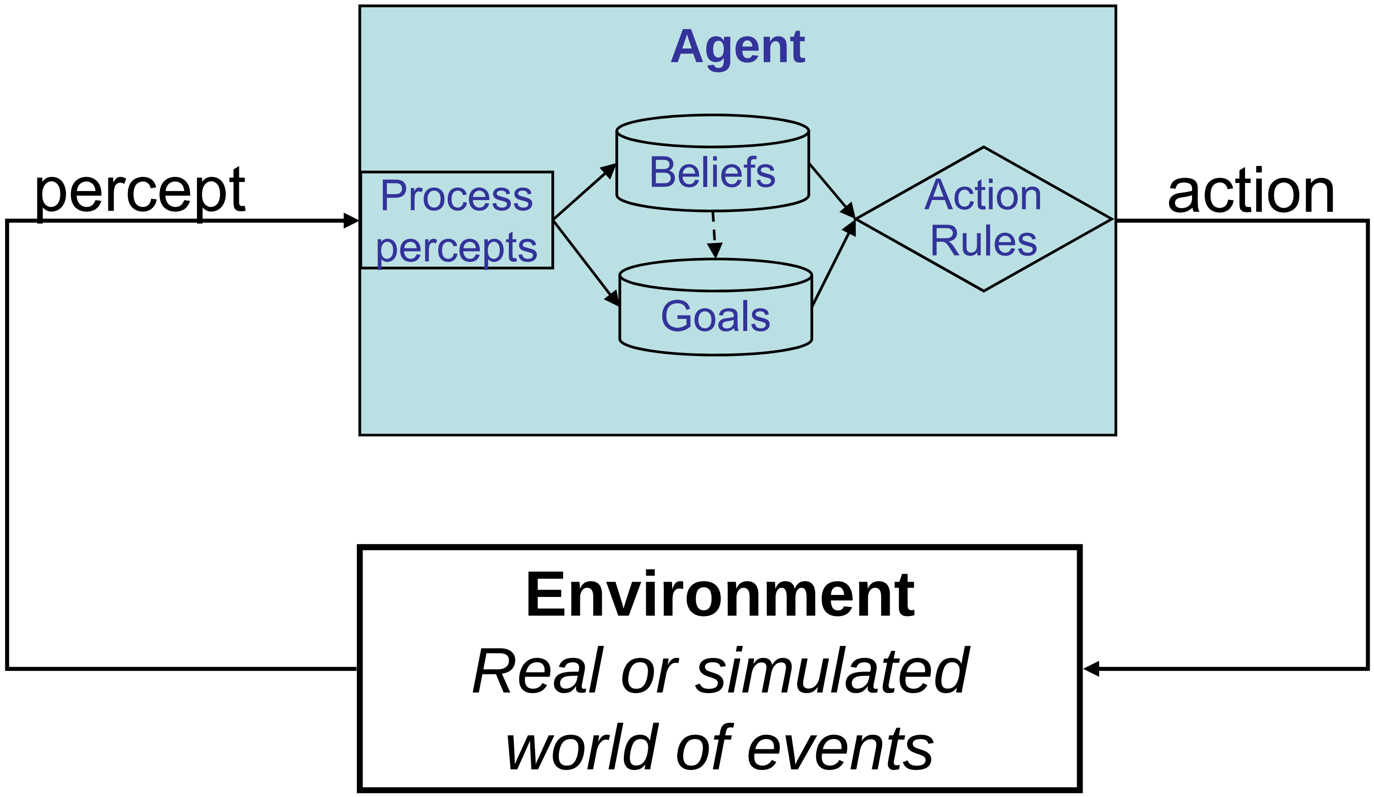
Some Architectural Considerations

Generic BDI Architecture

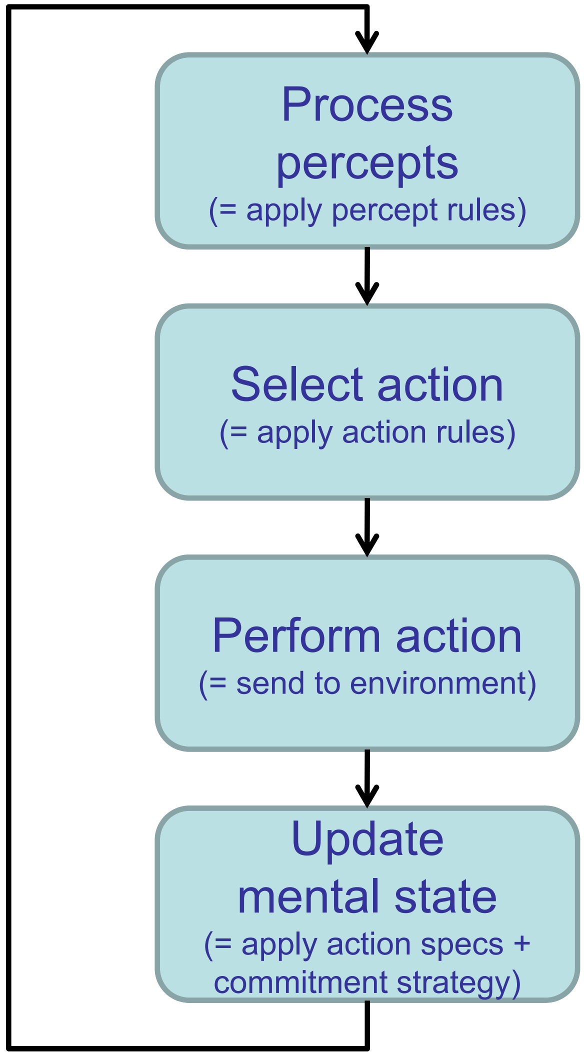



AgentSpeak(L) Agent
Beliefs
Belief Base
Beliefs
Percepts
1
BRF
External Events
Events
Internal Events
Events
2
S_E
Plan Library
Selected Event
Plans
3
Unify Event
Relevant Plans
Beliefs
Beliefs
4
Unify Context
Applicable Plans
5
S_O
Intended Means
6
S_I
Selected Intention
7
Execute Intention
Action
Intentions
Update Intention
Intentions
Push Subplan
New Intention
New
New
...

GOAL Architecture



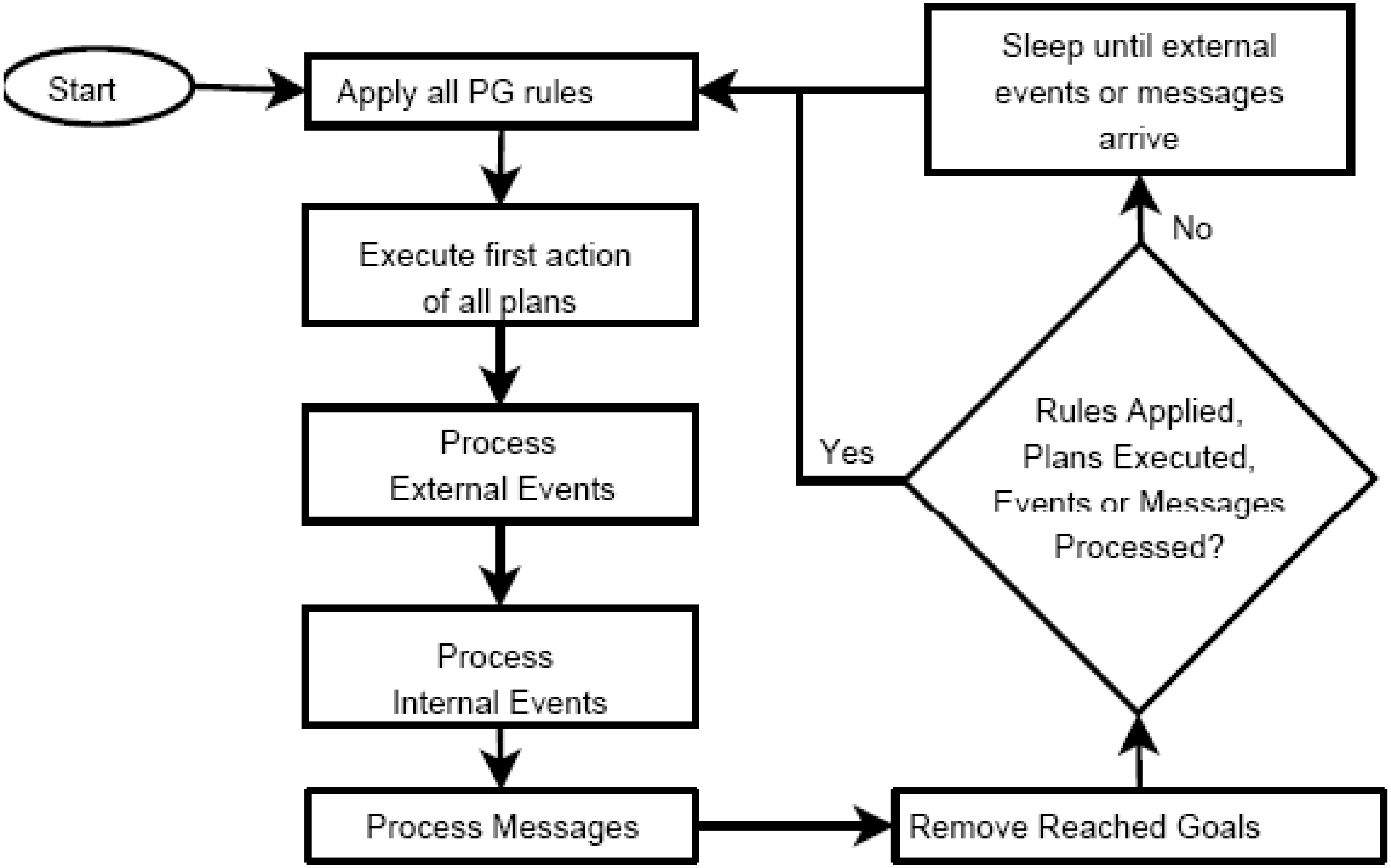
Interpreters: GOAL



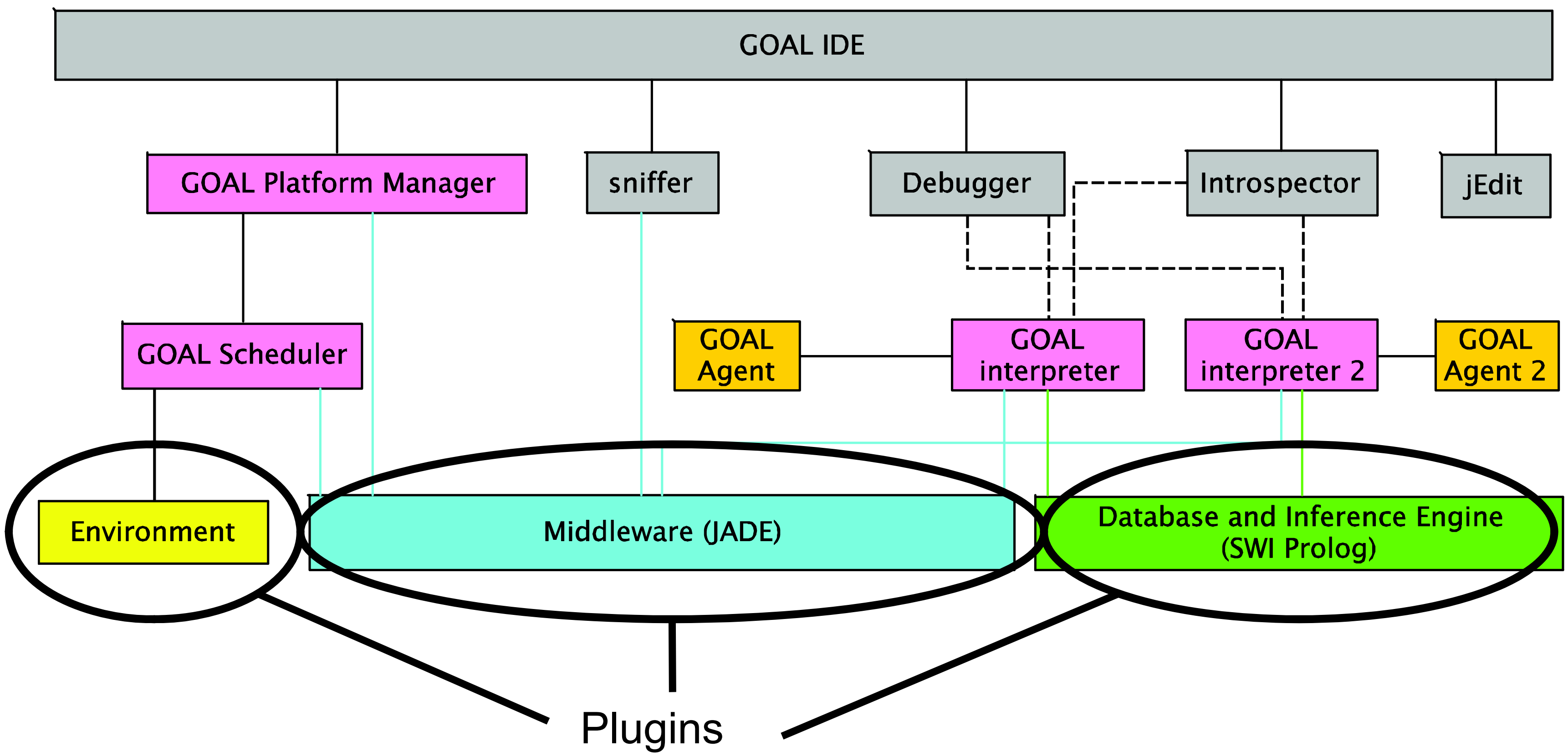
Also called **deliberation cycles**.

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

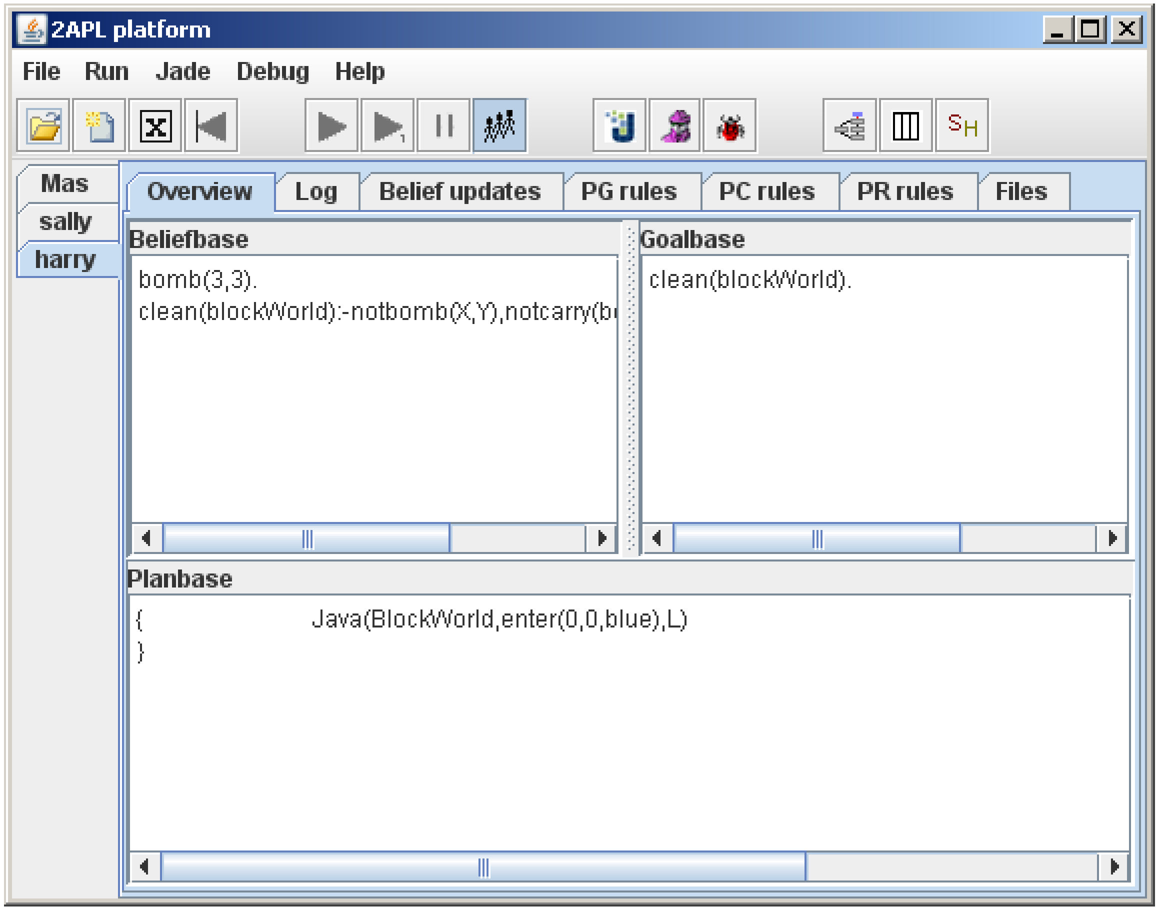
Interpreter: 2APL



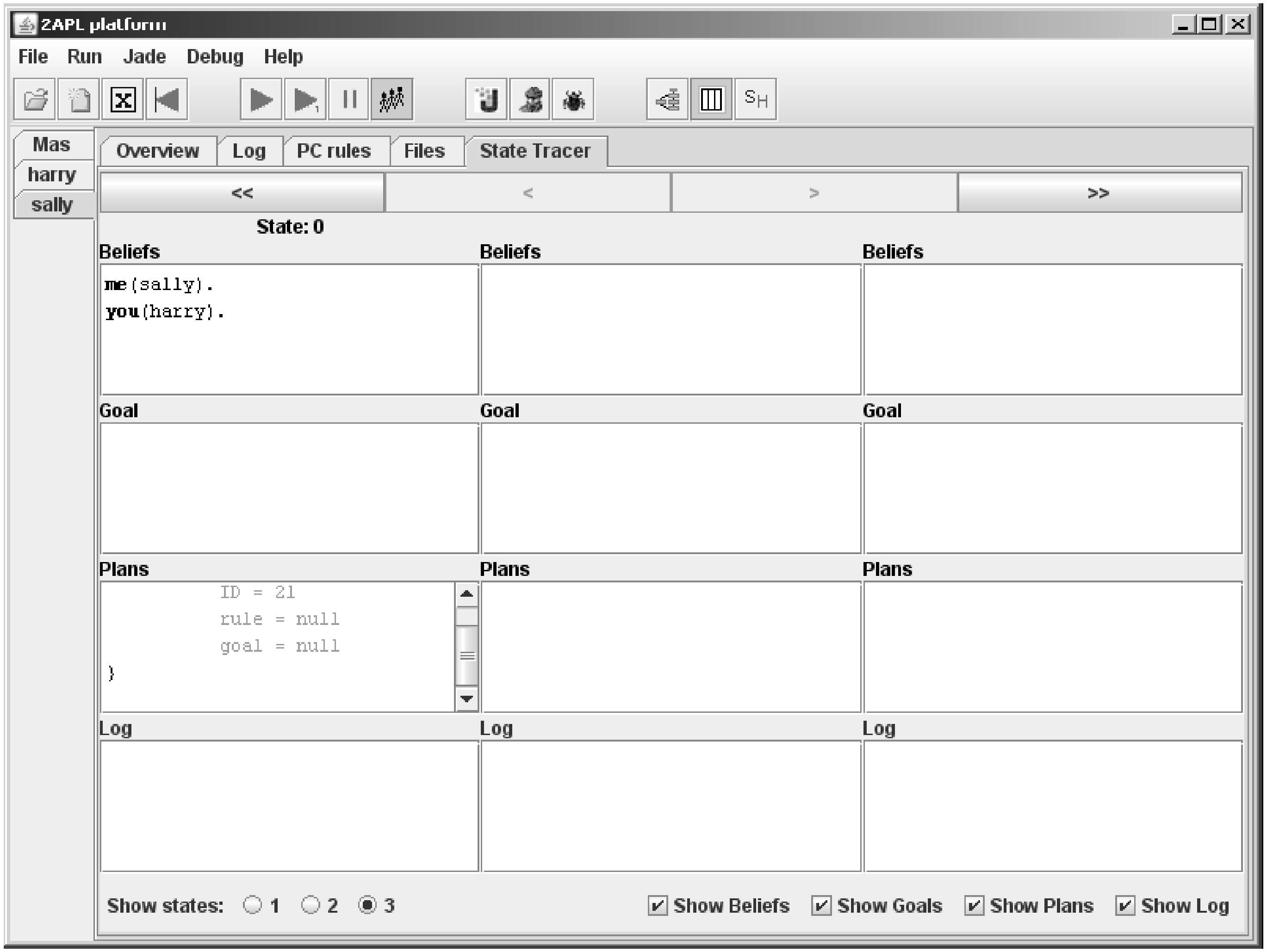
Under the Hood: Implementing AOP Example: GOAL Architecture



2APL IDE: Introspector



2APL IDE: State Tracer



Research Themes

A Personal Point of View

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?
- **Organizing MAS**: What are effective mas structures to organize communication, coordination, cooperation?
- 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

*** DOWNLOAD THESE SLIDES FROM THE GOAL WEBPAGE ***