Programming Intentional Agents in AgentSpeak(L) & *Jason*

Autonomous Systems
Sistemi Autonomi

Andrea Omicini *after Michele Piunti*
andrea.omicini@unibo.it

Dipartimento di Informatica – Scienza e Ingegneria (DISI)
Alma Mater Studiorum – Università di Bologna

Academic Year 2017/2018

1. Implementing BDI Architectures
2. AgentSpeak(L)
3. *Jason*
4. Conclusions

Next in Line...

BDI Abstract Control Loop

Agent control loop (v. 3) [Rao and Georgeff, 1995]

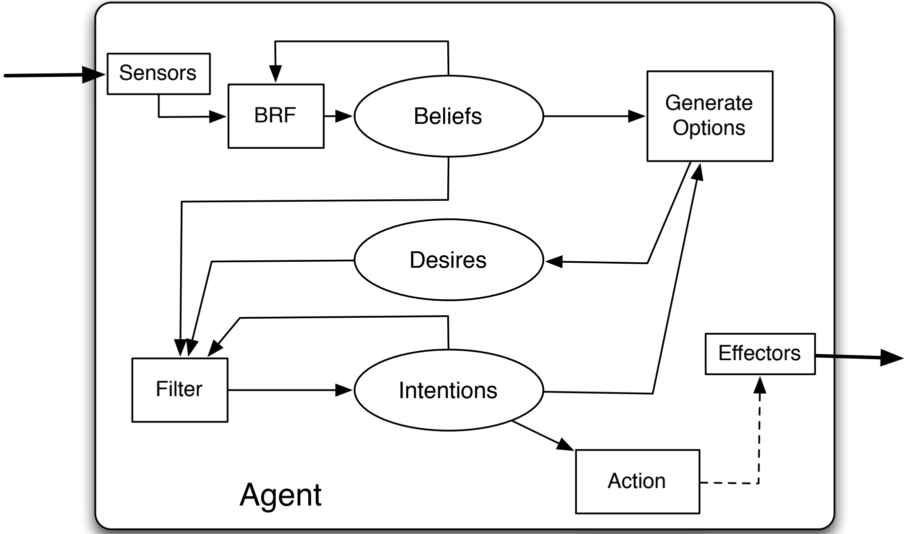
```
initialize-state();
while true do
      options := option-generator(event-queue);
      selected-options := deliberate(options);
      update-intentions(selected-options);
      execute();
      get-new-external-events();
      drop-successful-attitudes();
      drop-impossible-attitudes();
end-while
```

Structure of BDI Systems

BDI architectures are based on the following constructs

1. a set of beliefs
2. a set of desires (or goals)
3. a set of intentions
   - or better, a subset of the goals with an associated stack of plans for achieving them; these are the intended actions
4. a set of internal events
   - elicited by a belief change (i.e., updates, addition, deletion) or by goal events (i.e. a goal achievement, or a new goal adoption)
5. a set of external events
   - perceptive events coming form the interaction with external entities (i.e. message arrival, signals, etc.)
6. a plan library (repertoire of actions) as a further (static) component

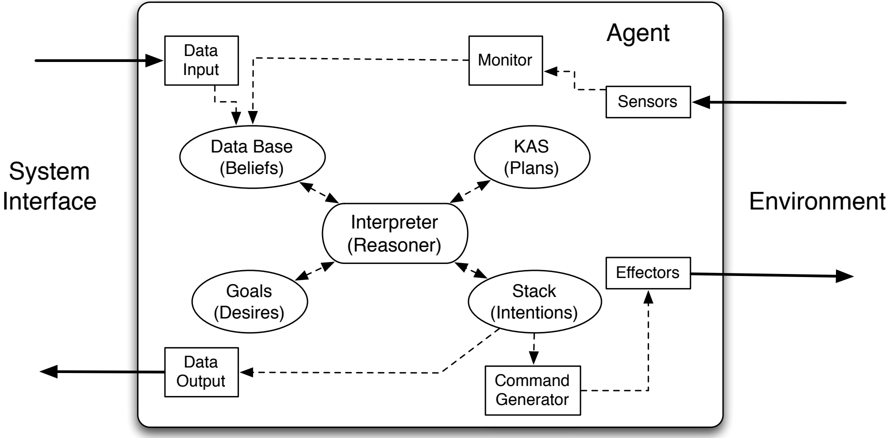
Basic Architecture of a BDI Agent [Wooldridge, 2002]



Procedural Reasoning System (PRS)

- PRS is one of the first BDI architectures [Georgeff and Lansky, 1987]
- PRS is a *goal-directed* and *reactive* planning system
    - goal-directedness allows reasoning about / performing complex tasks
    - reactiveness allows handling real-time behaviour in dynamic environments
- PRS is applied for high-level reasoning of robot, airport traffic control systems etc.

PRS Architecture



Next in Line. . .

AgentSpeak(L)

AgentSpeak(L)...

- is an *abstract language* used for describing and programming BDI agents
- inspired by PRS, dMARS [d'Inverno et al., 1998], and BDI Logics [Rao and Georgeff, 1995]
- originally proposed by Anand S. Rao [Rao, 1996]
- extended so as to make it a practical agent programming language [Bordini and Hübner, 2006]
- programs can be executed by the *Jason* platform [Bordini et al., 2007]
- has an operational semantics for extensions of AgentSpeak(L) providing a computational semantics for BDI concepts

Focus on. . .

Syntax of AgentSpeak(L)

- the main language constructs of AgentSpeak are
  - beliefs — current state of the agent, information about environment, and other agents
  - goals — state the agent desire to achieve and about which he brings about (Practical Reasoning) based on internal and external stimuli
  - plans — recipes of procedural means the agent has to change the world and achieve his goals
- the architecture of an AgentSpeak agent has four main components
  1. belief base
  2. plan library
  3. set of events
  4. set of intentions

Beliefs and Goals

Beliefs

beliefs if b is a predicate symbol, and $t_1, \ldots, t_n$ are (first-order) terms, $b(t_1, \ldots, t_n)$ is a *belief atom*

- ground belief atoms are *base beliefs*
- if Φ is a belief atom, Φ and $\neg\Phi$ are belief literals

Goals

goals If g is a predicate symbol, and $t_1, \ldots, t_n$ are terms, $!g(t_1, \ldots, t_n)$ and $?g(t_1, \ldots, t_n)$ are goals

1. '!' means Achievement Goals *(Goal to do)*
2. '?' means Test Goals *(Goal to know)*

Events I

- events occur as a consequence of changes in the agent's belief base or goal states
- events may signal to the agent that some situation is requiring servicing (*triggering events*)
- the agent indeed is supposed to react to such events by finding a suitable plan(s)
- due to events and goal processing, AgentSpeak(L) architectures are both
  - reactive
  - proactive

Events II

Events

events If $b(t)$ is a belief atom, $!g(t)$ and $?g(t)$ are goals, then $+b(t), -b(t), +!g(t), +?g(t), -!g(t)$, and $-?g(t)$ are *triggering events*

- let Φ be a literal, then the AgentSpeak triggering events are the following

$+\Phi$	Belief addition
$-\Phi$	Belief deletion
$+!\Phi$	Achievement-goal addition
$-!\Phi$	Achievement-goal deletion
$+?\Phi$	Test-goal addition
$-?\Phi$	Test-goal deletion

Plans I

- plans are recipes for achieving goals
- plans declaratively define a workflow of actions
- plans come along with the triggering and the context conditions that must hold in order to initiate the execution
- plans represent agent's means to achieve goals (their know-how)

Plans

plans if e is a triggering event, $b_1, \ldots, b_n$ are belief literals (plan context), and $h_1, \ldots, h_n$ are goals or actions (plan body), then
$e : b1 \wedge \ldots \wedge bn \leftarrow h_1; \ldots; h_n$
is a *plan* (where $e : c$ is called the plan's head)

Plans II

PlanBody

Let Φ be a literal, then the PlanBody (i.e., intentions in AgentSpeak) can include the following elements:

- $!\Phi$ achievement goals
- $?\Phi$ test goals
- $+\Phi$ belief addition
- $-\Phi$ belief deletion
- Φ actions
- $.\Phi$ internal actions *(not actually here, this is Jason. . .)*

Plans III

General structure of an AgentSpeak plan

```
triggering_event: context <- body.
```

- the `triggering_event` denotes the events that the plan is meant to handle
- the `context` represents the circumstances in which the plan can be used
  - logical expression, typically a conjunction of literals to be checked whether they follow from the current state of the belief base (Belief Formulae)
- the `body` is the course of action to be used to handle the event if the context is believed true at the time a plan is being chosen to handle the event
  - a sequence of actions and (sub) goals to achieve that goal

AgentSpeak(L) Examples

```
/* Initial Beliefs */
likes(radiohead).
phone_number(covo,"05112345")

/* Belief addition */
+concert(Artist, Date, Venue)
  : likes(Artist)
  <-   !book_tickets(Artist, Date, Venue).

/* Plan to book tickets */
+!book_tickets(A,D,V)
  : not busy(phone)
  <-   ?phone_number(V,N);   /* Test Goal to Retrieve a Belief */
       !call(N);
       . . .;
       !choose seats(A,D,V).
```

Focus on. . .

AgentSpeak(L) Semantics I

AgentSpeak(L) has an operational semantics defined in terms of agent configuration $\langle B, P, E, A, I, S_e, S_o, S_I \rangle$, where

- B is a set of beliefs
- P is a set of plans
- E is a set of events (external and internal)
- A is a set of actions that can be performed in the environment
- I is a set of intentions each of which is a stack of partially instantiated plans
- S_e, S_o, S_I are selection functions for events, options, and intentions

AgentSpeak(L) Semantics II

The selection functions

S_e selects an *event* from E. The set of events is generated either by requests from users, from observing the environment, or by executing an intention

S_o selects an *option* from P for a given event. An option is an *applicable plan* for an event, i.e. a plan whose triggering event is unifiable with event and whose condition is derivable from the belief base

S_I selects an *intention* from I to execute

Semantics of Intention Execution

Semantics of intention execution

- $tr : ct \leftarrow +\varphi; \ldots \Rightarrow$ generates event $+\varphi$ and updates beliefs. If there is no applicable plan for $+\varphi$, discard the event.
- $tr : ct \leftarrow -\varphi; \ldots \Rightarrow$ generates event $-\varphi$ and updates beliefs. If there is no applicable plan for $-\varphi$, discard the event.
- $tr : ct \leftarrow !\varphi; \ldots \Rightarrow$ generates event $+!\varphi$. If there is no applicable plan for $+!\varphi$, remove plan and generate $-!\psi$ if $tr = +!\psi$ (or $-?\psi$ if $tr = +?\psi$).
- $tr : ct \leftarrow ?\varphi; \ldots \Rightarrow$ generates event $+?\varphi$ If there is no applicable plan for $+?\varphi$, remove plan and generate $-!\psi$ if $tr = +!\psi$ (or $-?\psi$ if $tr = +?\psi$).
- $tr : ct \leftarrow \varphi; \ldots \Rightarrow$ if the action fails, remove plan and generate $-!\psi$ if $tr = +!\psi$ (or $-?\psi$ if $tr = +?\psi$).
- $tr : ct \leftarrow .\varphi; \ldots \Rightarrow$ if the internal action fails, remove plan and generate $-!\psi$ if $tr = +!\psi$ (or $-?\psi$ if $tr = +?\psi$).

If no plan is applicable for a generated $-!\psi$ or $-?\psi$, then the whole intention is disregarded and an error message is printed

Agent Configuration

Configuration of an AgentSpeak agent

$$\langle ag, C, M, T, s \rangle$$

- *ag* is an AgentSpeak *program* consisting of a set of beliefs and plans
- $C = \langle I, E, A \rangle$ is the *agent circumstance*
- $M = \langle In, Out, SI \rangle$ is the *communication component*
- $T = \langle R, Ap, \iota, \varepsilon, \rho \rangle$ is the *temporary information component*
- *s* is the current step within an agent's *reasoning cycle*

Circumstance Component

$$\langle ag, C, M, T, s \rangle$$

Agent's circumstance

$$C = \langle I, E, A \rangle$$

- I is a set of intentions $\{i, i', \ldots\}$; each intention i is a stack of partially instantiated plans
- E is a set of events $\{(tr, i), (tr', i'), \ldots\}$; each event is a pair (tr, i), where tr is a triggering event and i is an intention (a stack of plans in case of an internal event or $\top$ representing an external event)
- A is a set of actions to be performed in the environment; an action expression included in this set tells other architecture components to actually perform the respective action on the environment, thus changing it.

Communication Component

$$\langle ag, C, M, T, s \rangle$$

Agent communication

$$M = \langle In, Out, SI \rangle$$

- *In* is the mail inbox: the system includes all messages addressed to this agent in this set
- *Out* is where the agent posts all messages it wishes to send to other agents
- *SI* is used to keep track of intentions that were suspended due to the processing of communication messages

Communication Component

$$\langle ag, C, M, T, s \rangle$$

Agent communication

$$M = \langle In, Out, SI \rangle$$

- *In* is the mail inbox: the system includes all messages addressed to this agent in this set
- *Out* is where the agent posts all messages it wishes to send to other agents
- *SI* is used to keep track of intentions that were suspended due to the processing of communication messages

Message

$$\langle messageid, agentid, ilf, content \rangle$$

Temporary Information Component

$$\langle ag, C, M, T, s \rangle$$

Temporary information

$$T = \langle R, Ap, \iota, \varepsilon, \rho \rangle$$

- R for the set of relevant plans (for the event being handled)
- Ap for the set of applicable plans (the relevant plans whose context are true)
- ι, ε and ρ keep record of a particular intention, event and applicable plan (respectively) being considered along the execution of an agent

Deliberation Steps

The current step s within an agent's reasoning cycle is one of the following elements:

- **ProcMsg** processing a message from the agent's mail inbox
- **SelEv** selecting an event from the set of events
- **RelPl** retrieving all relevant plans
- **ApplPl** checking which of those are applicable
- **SelAppl** selecting one particular applicable plan (the intended means)
- **AddIM** adding the new intended means to the set of intentions
- **SelInt** selecting an intention
- **ExecInt** executing the select intention
- **ClrInt** clearing an intention or intended means that may have finished in the previous step

Next in Line. . .

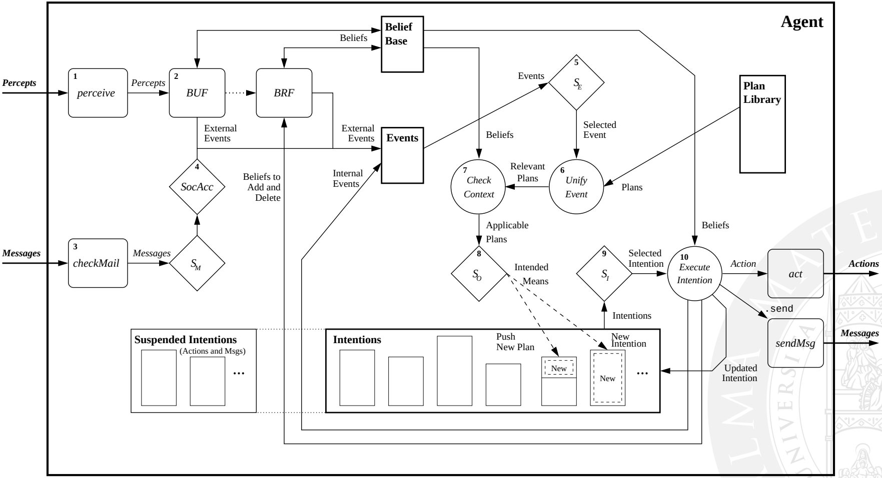
Focus on. . .

Jason [Bordini et al., 2007]

- developed by Jomi F. Hübner and Rafael H. Bordini
- *Jason* implements the operational semantics of a variant of AgentSpeak [Bordini and Hübner, 2006]
- extends AgentSpeak, which is meant to be the language for defining agents
- adds a set of powerful mechanism to improve agent abilities
- extensions aimed at a more practical programming language
  - High level language to define agents (goal oriented) behaviour
  - Java as low level language to realise mechanisms (i.e. agent internal functions) and customise the architecture
- comes with a framework for developing multi-agent systems [1]

[1] http://jason.sourceforge.net/

Jason Architecture



Jason Reasoning Cycle

1. perceiving the environment
2. updating the belief base
3. receiving communication from other agents
4. selecting 'socially acceptable' messages
5. selecting an event
6. retrieving all relevant plans
7. determining the applicable plans
8. selecting one applicable plan
9. selecting an intention for further execution
10. executing one step of an intention

jason.asSemantics.TransitionSystem

```
public void reasoningCycle() {
     try {
         C.reset();   //C is actual Circumstance
         if (nrcslbr >= setts.nrcbp()) {
             nrcslbr = 0;
             ag.buf(agArch.perceive());
             agArch.checkMail();
         }
         nrcslbr++;       // counting number of cycles
         if (canSleep()) {
             if (ag.pl.getIdlePlans() != null) {
                 logger.fine("generating idle event");
                 C.addExternalEv(PlanLibrary.TE_IDLE);
             } else {
                 agArch.sleep();
                 return;
         }     }
         step = State.StartRC;
         do {
             if (!agArch.isRunning()) return;
             applySemanticRule();
         } while (step != State.StartRC);
         ActionExec action = C.getAction();
         if (action != null) {
             C.getPendingActions().put(action.getIntention().getId(), action);
             agArch.act(action, C.getFeedbackActions());
         }
     } catch (Exception e) {
         conf.C.create();  //ERROR in the transition system, creating a new C
     }
}
```

Focus on. . .

Jason as an Agent Programming Language

- *Jason* include all the syntax and the semantics already defined for AgentSpeak
- boolean operators
  - `==, <, <=, >, >=, &, |, \==, not`
- arithmetic
  - `+, -, /, *, **, mod, div`
- then, *Jason* includes several extesions
- e.g.: let Φ be a literal, then a *Jason* PlanBody can include the following additional elements:
  - $!!\Phi$ to launch a given plan Φ as a new intention (the new intention will not be related to the current one, its execution will be *as if* it is in a new thread)
  - $-+\Phi$ to update a Belief Φ in an atomic fashion (atomic deletion and update)

Belief Annotations

Jason introduces the notion of *annotated predicates*:

$$p_s(t_1, ..., t_n)[a_1, ..., a_m]$$

where a_i are first order terms

- all predicates in the belief base have a special annotation $source(s_i)$ where $s_i \in \{self, percept\} \cup AgId$
  - `myLocation(6,5)[source(self)].`
  - `red(box1)[source(percept)].`
  - `blue(box1)[source(ag1)].`
- agent developer can define customised predicates (i.e. grade of certainty on that belief)
  - `colourblind(ag1)[source(self),doc(0.7)].`
  - `lier(ag1)[source(self),doc(0.2)].`

Strong Negation

- strong negation (operator $\sim$) is another *Jason* extension to AgentSpeak
- to allow both closed-world and open-world assumptions

```
+!pit_stop(fuel(T),  tires(_))
  :  not raining & not ~raining   /* Lack of knowledge:
               there is no belief indicating raining
               neither belief indicating  ~raining */
  <- -+tires(intermediate);        /* Atomic Belief Update */
     !fuel(T+2);
     ...
+!pit_stop(fuel(T),  tires(_))
  : raining  /* There is a belief indicating raining */
  <- -+tires(rain);   /* Atomic Belief Update */
     !fuel(T+5);
     ...

+!pit_stop(fuel(T),  tires(_))
  : ~raining  /* There is a belief indicating ~raining */
  <- -+tires(slick);   /* Atomic Belief Update */
     !fuel(T);
     ...
```

Belief Rules

In *Jason*, beliefs (and their annotations) can be pre-processed with Prolog-like rules:

```
likely_color(Obj,C)
  :- colour(Obj,C)[degOfCert(D1)]
     & not (
        colour(Obj,_)[degOfCert(D2)]
        & D2 > D1 )
     & not ~colour(Obj,B).
```

Handling Plan Failures

Handling plan failures is very important when agents are situated in dynamic and non-deterministic environments

- goal-deletion events are another *Jason* extension to AgentSpeak
- −!g
- to create an agent that is blindly committed to goal g:

```
+!g(X) : goalstate
    <- true.
+!g(X) : not goalstate
    <- ...
        ?g.
...
-!g : true  /* Goal deletion event */
    <- !g.
```

Plan Annotations

Plan can have annotations too (e.g., to specify meta-leval information)

- selection functions (Java) can use such information in plan/intention selection
- possible to change those annotations dynamically (e.g., to update priorities)
- annotations go in the plan label

```
@aPlan[ chance_of_success(0.3), usual_payoff(0.9),
  any_other_property]
+!g(X) :  c(t)
  <- a(X).
```

- (`chanche_of_success * usual_payoff`) is the expected utility for that plan

Internal Actions

- in *Jason* plans can contain an additional structure: *internal action* .Φ
- self-contained actions which code is packed and atomically executed as part of the agent reasoning cycle
- internal actions can be used for special purpose activities
  - to interact with Java objects
  - to invoke legacy systems elegantly
  - as we will see in the rest of the course, to use *artifacts* in A&A systems
- example of user defined internal action:

  `userLibrary.userAction(X,Y,R)`

  can be used to manipulate parameters X, Y and unify the result of that manipulation in R

Defining New Internal Actions

Internal action: `myLib.randomInt(M, N)` unifies `N` with a random int between 0 and `M`.

```
package myLib;

import jason.JasonException;
import jason.asSemantics.*;
import jason.asSyntax.*;

public class randomInt extends DefaultInternalAction {

    private java.util.Random random = new java.util.Random();

    @Override
    public Object execute(TransitionSystem ts, Unifier un, Term[] args) throws Exception {
        if (!args[0].isNumeric() ||  !args[1].isVar())
                throw new JasonException("check arguments");
        try {
            int R = random.nextInt( ((numberTerm)args[0]).solve() );
            return
                 un.unifies(args[1], new NumberTermImpl(R));
        } catch (Exception e) {
            throw new JasonException("Error in internal action 'randomInt'", e);
        }
    }
}
```

Predefined Internal Actions

- many internal actions are available for: printing, sorting, list/string operations, manipulating the beliefs/annotations/plan library, waiting/generating events, etc. (see *jason.stdlib*)
- predefined internal actions have an empty library name

.print(1,X,"bla") prints out to the console the concatenation of the string representations of the number 1, of the value of variable X, and the string "bla"

.union(S1,S2,S3) $S3$ is the union of the sets S1 and S2 (represented by lists). The result set is sorted

.desire(D) checks whether D is a desire: D is a desire either if there is an event with $+!D$ as triggering event or it is a goal in one of the agent's intentions

.intend(I) checks if I is an intention: I is an intention if there is a triggering event $+!I$ in any plan within an intention; just note that intentions can be suspended and appear in E, PA, and PI as well

.drop_desire(I) removes events that are goal additions with a literal that unifies with the one given as parameter

.drop_intention(I) drops all intentions which would make *.intend* true

Internal Actions used for Message Passing

sender agent A sends a message to agent B using a special internal action:

```
.send(B, ilf, m(X))
.broadcast(ilf, m(X))
```

- B is the unique name of the agent that will receive the message (or a list of names)
- $ilf \in \{tell, untell, achieve, unachieve, askOne, askAll, askHow, tellHow, untellHow\}$
- $m(X)$ the content of the message

receiver agent B receives the message from A as a triggering event

- Handles it by customizing a reaction:

```
+m(X)[source(A)] : true
<- dosomething;...
```

Environments

- to build and deploy a MAS you need to rely on some sort of environment where the agents are situated
- the environment has to be designed (and implemented as well)
- there are two ways to do this:
  1. defining perceptions and actions so to operate on specific environments
     - this is done defining in Java lower-level mechanisms, and by specialising the Agent Architecture and Agent classes (see later)
  2. creating a 'simulated' environment
     - this is done in Java by extending *Jason*'s Environment class and using methods such as `addPercept(String Agent, Literal Percept)`

Example of an Environment Class

```
import jason.*;
import ...;
public class myEnv extends Environment{
....
   public myEnv() {
      Literal loc = Literal.parseLiteral("location(3,5)");
      addPercept(pos1);
   }

   public boolean executeAction(String ag, Term action)  {
      if (action.equals(...))  {
         addPercept(ag,
                    Literal.parseLiteral("location(souffle,c(3,4))");
      }
      ...
      return true;
   }
}
```

Focus on...

Hierarchical Planning I

- hierarchical abstraction is a well-known principle
- exhibits a great effectiveness in planning
- used to reduce a composite intention – or a given task – to a greater number of independent sub-intentions – or sub-tasks – placed at a lower level of abstraction
- an agent can manage at runtime an alternating hierarchy of (meta)goals and plans, which emerge from top-level goals over plans to subgoals and so forth
  - this highly simplifies the structure of plans
  - allow the plans to be conceived around self-contained actions (the leafs of the goal hierarchy) which can be reused with different purposes too
- defined having in mind the problem domain (the goal to be achieved) and trying to immagine those fine grained actions which in turn are supposed to accomplish the required activities

Hierarchical Planning II

- differently from traditional planning systems, which mainly make an *offline* planning, Intentional Systems need to plan in dynamic environments and need to cope changing contexts and situations [Sardina et al., 2006]

planning systems is offline — can create plans to achieve goals by composing actions in repertoire

BDI planning hybrid approach — the plans are defined at design time and at the language level *but* their execution is ruled by the architecture (means ends reasoning) according to context conditions (i.e., *Jason*, Jadex) or planning rules (i.e., 2APL).

Next in Line. . .

Conclusions

AgentSpeak
- goal-oriented notion of agency
- mentalistic notions as building blocks
- agent programming
- logic + BDI
- operational semantics

Jason
- AgentSpeak interpreter
- implements the operational semantics
- support for Agent Comunication Language
- highly customisable, open source

References I

Bordini, R. H. and Hübner, J. F. (2006).
BDI agent programming in AgentSpeak using *Jason* (tutorial paper).
In Toni, F. and Torroni, P., editors, *Computational Logic in Multi-Agent Systems*, volume 3900 of *Lecture Notes in Computer Science*, pages 143–164. Springer.
6th International Workshop, CLIMA VI, London, UK, June 27-29, 2005. Revised Selected and Invited Papers.

Bordini, R. H., Hübner, J. F., and Wooldridge, M. J. (2007).
Programming Multi-Agent Systems in AgentSpeak using Jason.
John Wiley & Sons, Ltd.
Hardcover.

d'Inverno, M., Kinny, D., Luck, M., and Wooldridge, M. (1998).
A formal specification of dMARS.
In *Intelligent Agents IV Agent Theories, Architectures, and Languages*, volume 1365 of *Lecture Notes in Computer Science*, pages 155–176. Springer Berlin Heidelberg.
4th International Workshop, ATAL'97 Providence, Rhode Island, USA, July 24–26, 1997 Proceedings.

References II

Georgeff, M. P. and Lansky, A. L. (1987).
Reactive reasoning and planning.
In Forbus, K. and Shrobe, H., editors, *6th National Conference on Artificial Intelligence (AAAI-87)*, volume 2, pages 677–682, Seattle, WA, USA. AAAI Press.
Proceedings.

Rao, A. S. (1996).
AgentSpeak(L): BDI agents speak out in a logical computable language.
In Van de Velde, W. and Perram, J. W., editors, *Agents Breaking Away*, volume 1038 of *LNCS*, pages 42–55. Springer.
7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW'96), Eindhoven, The Netherlands, 22-25 January 1996, Proceedings.

Rao, A. S. and Georgeff, M. P. (1995).
BDI agents: From theory to practice.
In Lesser, V. R. and Gasser, L., editors, *1st International Conference on Multi Agent Systems (ICMAS 1995)*, pages 312–319, San Francisco, CA, USA. The MIT Press.

References III

Sardina, S., de Silva, L., and Padgham, L. (2006).
Hierarchical planning in BDI agent programming languages: A formal approach.
In Stone, P. and Weiss, G., editors, *5th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS '06)*, pages 1001–1008, Hakodate, Japan. ACM.

Wooldridge, M. J. (2002).
An Introduction to MultiAgent Systems.
John Wiley & Sons Ltd., Chichester, UK.

Programming Intentional Agents in AgentSpeak(L) & *Jason*

Autonomous Systems
Sistemi Autonomi

Andrea Omicini *after Michele Piunti*
andrea.omicini@unibo.it

Dipartimento di Informatica – Scienza e Ingegneria (DISI)
Alma Mater Studiorum – Università di Bologna

Academic Year 2017/2018