Agent Architectures and Programming

Michal Pechoucek, Branislav Bošanský & Michal Jakob

AE4M36MAS Autumn 2016

Selected illustrations taken Russel and Norvig – Artificial Intelligence: Modern Approach
Selected slides have been prepared by Terry Payne, University of Liverpool, who kindly provided them for OI/MAS

Where are we?

Agent architectures (inc. BDI architecture)

Logics for MAS

Non-cooperative game theory

Coalition game theory

Mechanism design

Auctions

Social choice

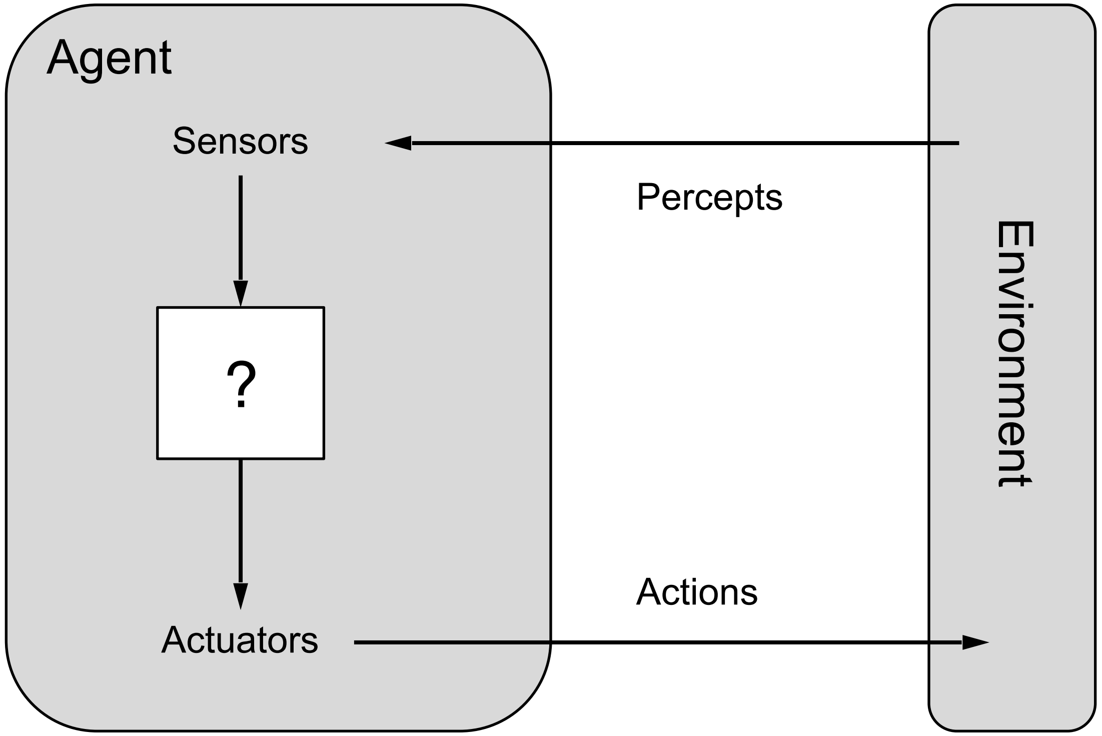
Distributed constraint reasoning
(satisfaction and optimization)

Introduction to Agents

Agent Architectures

Implementing the Agent

How should one implement the agent function?



Concern 1: Rationality

Concern 2: Computability and tractability

Hierarchy of Agents

The key challenge for AI is to find out how to write programs that produce rational behaviour from a small amount of code rather than from a large number of table entries.

4+1 basic types of agents in the order of increasing capability:

1. simple reactive agents
2. model-based agents with state
3. goal-based agents
4. utility-based agents
5. learning agents

There is a link between the complexity of the task and the minimum agent architecture required to implement a rational agent.

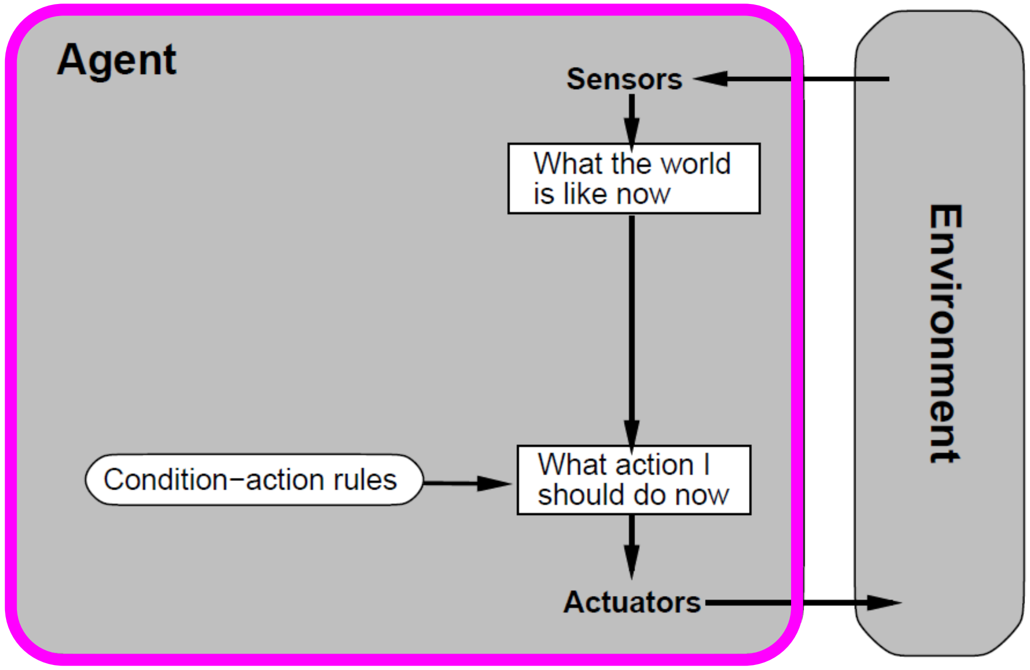
Running Example: Robotic Taxi

Task specification

- Performance measure: the overall profit (= passenger revenues - fines)
- Environment: road network with traffic signs, passengers
- Actions (actuators): driving between junctions, picking up and dropping out passengers
- Percepts (sensors): current GPS location, junction layout, traffic signs, passengers

Simple Reactive Agents

Simple reactive/reflex agent chooses the next action on the basis of the current percept only.



Simple Reactive Agent

Condition-action rules provide a way to present common regularities appearing in input/output associations, *example:*

```
If car-in-front-is-braking
=> initialize-braking
```

```
Function SIMPLE-REACTIVE-AGENT(percept)
      rule <= RULE-MATCHING(rules)
      action <= rule.ACTION
      Return action
```

```
Function RULE-MATCHING(state, rules) ...
```

Simple Reactive Agent for Robotic Taxi

Simple program:

```
If a passenger at your location
=> pickup the passenger
else Continue in the left-most direction possible
```

More sophisticated program:

```
Turn-directions depend on the current GPS location (can
implement specific fixed route through the city)
```

Issues with Reactive Agents

Robotic taxi

- driving to a given destination
- respecting traffic signs (e.g. speed limits)
- getting stuck in loops

Reactive agents are simple but of limited intelligence, rational if

1. the environment is fully observable and
2. the decision can be made based solely on the current percept

otherwise may leads to suboptimal action choices, infinite loops.

Issues with Reactive Agents

Robotic taxi

- driving to a given destination
- respecting traffic signs (e.g. speed limits)
- getting stuck in loops

Reactive agents are simple but of limited intelligence, rational if

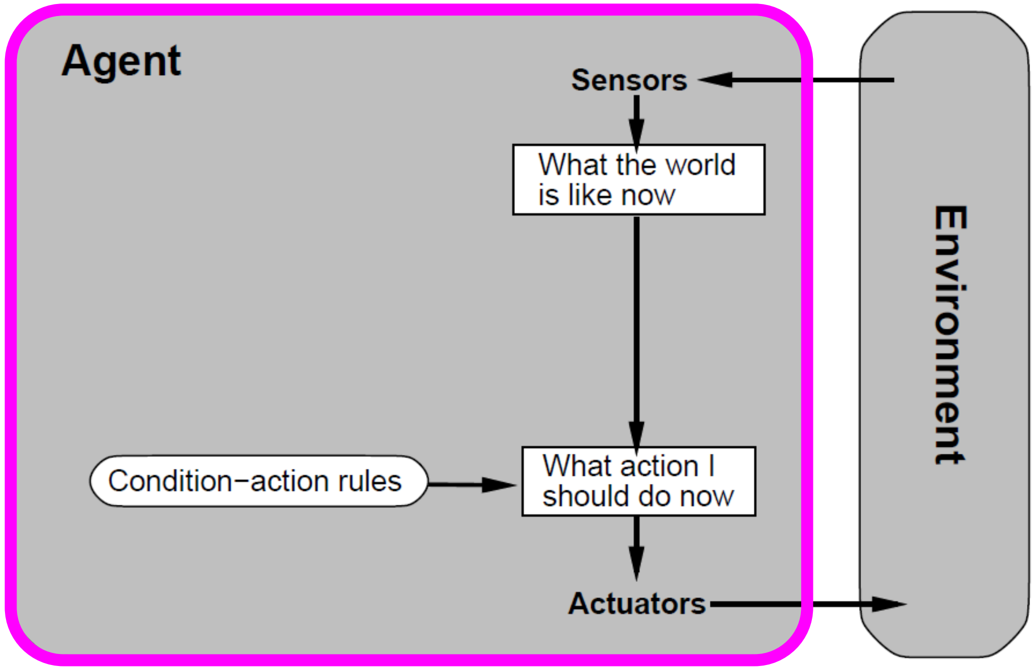
1. the environment is fully observable and
2. the decision can be made based solely on the current percept

otherwise may leads to suboptimal action choices, infinite loops.

$\rightarrow$ It can be advantageous
to **store information about the world** in the agent.

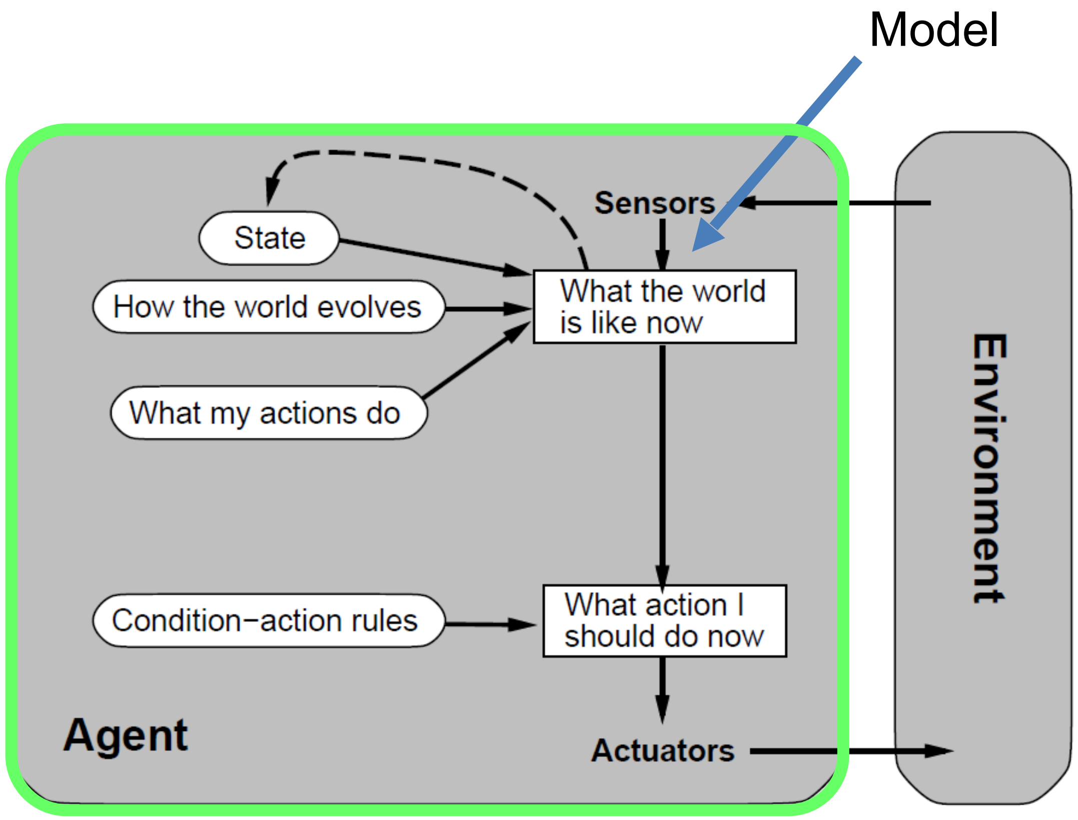
Model-based Agent

Keeps track of the world by extracting relevant information from percepts and storing it in its memory.



Model-based Agent

Keeps track of the world by extracting relevant information from percepts and storing it in its memory.



Model-based Agent

```
Function SIMPLE-REACTIVE-AGENT(percept)
      state <=
             UPDATE-STATE(state, action, percept, model)
      rule <= RULE-MATCHING(state, rules)
      action <= rule.ACTION
      Return action
```

States tracked in the model

- passengers' destinations
- traffic signs
- visited locations (to avoid cycles)
- pickup locations (=> learning)

Model-based Reactive Taxi Agent

States tracked in the model

- passengers’ destinations
- traffic signs
- visited locations (to avoid cycles)
- pickup locations (=> learning)

Issues with Model-based Agents

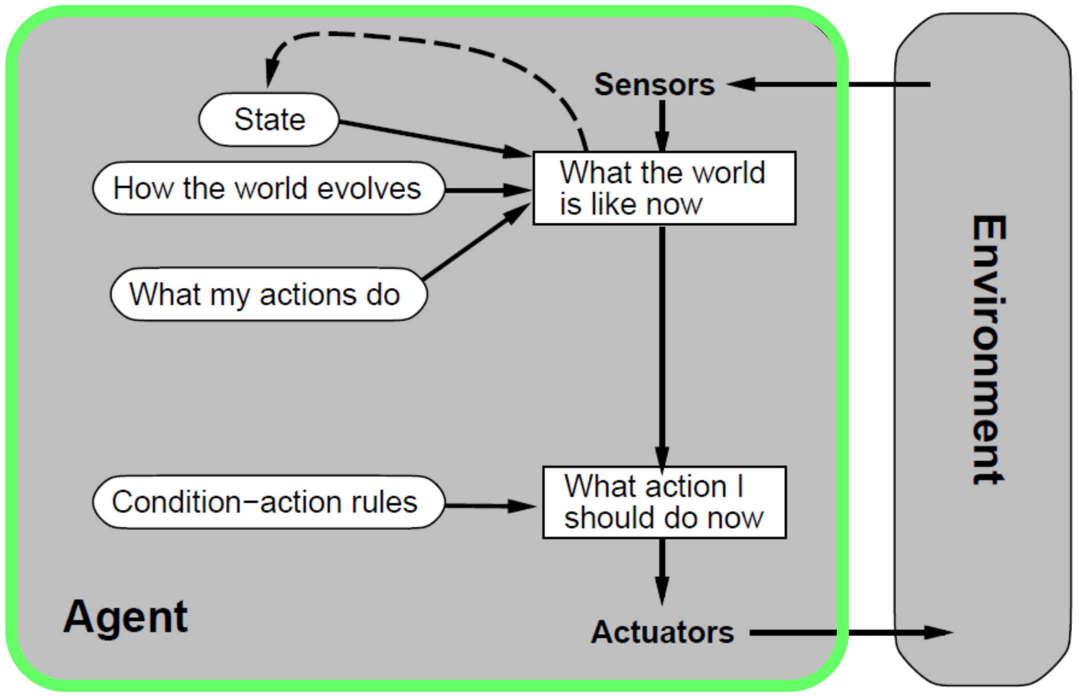
Taxi agent: Hot to get to a destination?

- Always move towards the destination location
  $\rightarrow$ can end-up in dead end streets
- Hard-code routes between all locations
  - » *memory demanding and of limited intelligence*
  - » *e.g. requires reprogramming the agent if street network changes*

Cause:

- *whats* and *hows* tightly coupled (impossible to tell the agent what to do)
- the agent does not anticipate the effects of its actions (only finds out the result after having executed the action)

Goal-based Agents

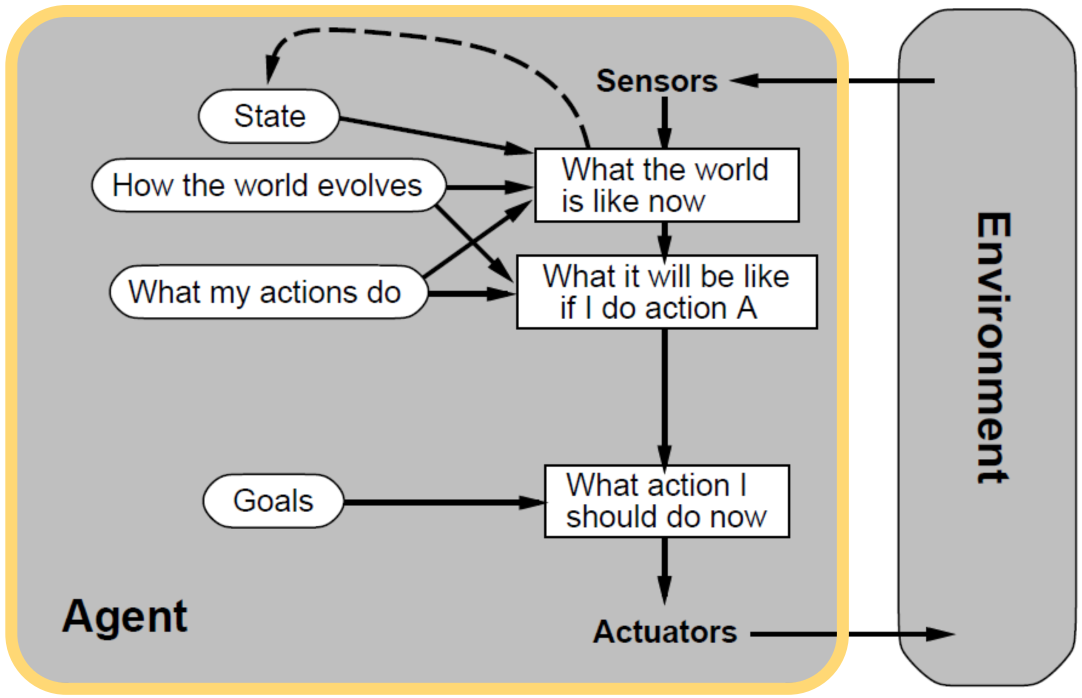


Goal-based agents are more flexible

Problem: goals are not necessarily achievable by a single action:

→ search and planning

Goal-based Agents


State
How the world evolves
What my actions do
Goals
Sensors
What the world is like now
What it will be like if I do action A
What action I should do now
Actuators
Agent
Environment

Goal-based agents are more flexible

Problem: goals are not necessarily achievable by a single action:

$\rightarrow$ search and planning

Goal-based Taxi Agent

Uses planning

- Uses a map to find a sequence of movement actions that brings the taxi to the destination reliable

Issue

- will not choose the fastest route
- will not balance revenue vs. fees/fines

Cause: goals alone are not sufficient for decision making:

1. there may be multiple ways of achieving them;
2. agents may have several conflicting goals that cannot be achieved simultaneously.

Utility-based Agents

Goals only a very crude (binary) distinction between “happy” and “unhappy” states.

We introduce the concept of utility:

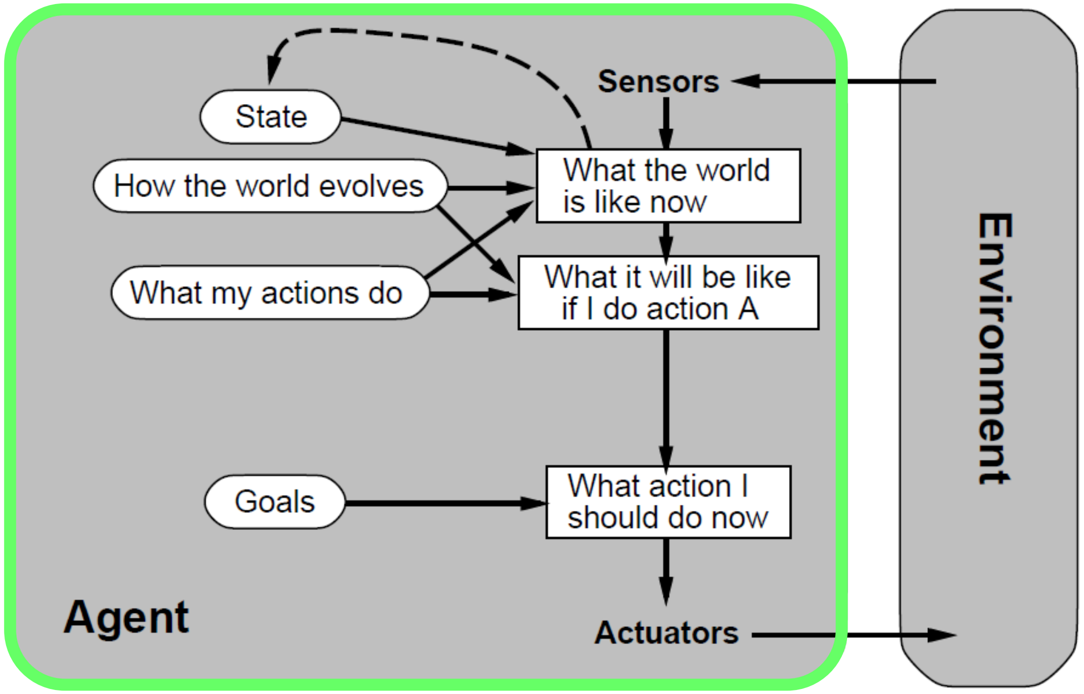
- utility is a function that maps a state onto a real number; it captures “quality” of a state
- if an agent prefers one world state to another state then the former state has higher utility for the agent.

Utility can be used for:

1. choosing the best plan
2. resolving conflicts among goals
3. estimating the successfulness of an agent if the outcomes of actions are uncertain.

Utility-based Agents

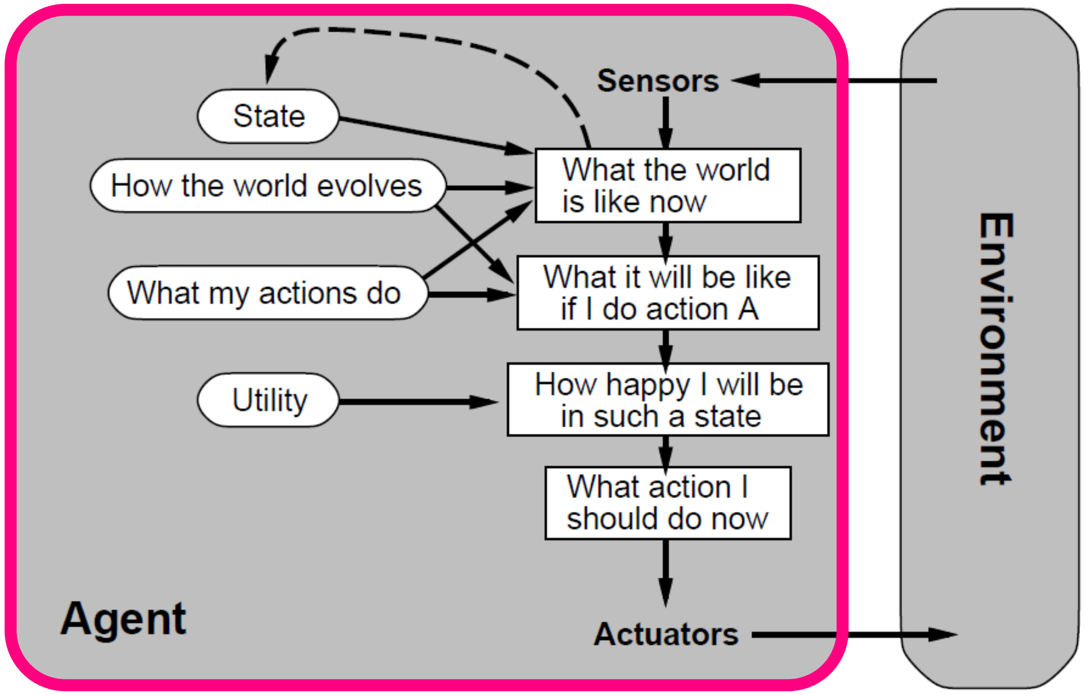
Utility-based agent use the utility function to choose the most desirable action/course of actions to take



Sensors
State
How the world evolves
What my actions do
What the world is like now
What it will be like if I do action A
Goals
What action I should do now
Agent
Actuators
Environment

Utility-based Agents

Utility-based agent use the utility function to choose the most desirable action/course of actions to take


Sensors
State
How the world evolves
What my actions do
Utility
What the world is like now
What it will be like if I do action A
How happy I will be in such a state
What action I should do now
Agent
Actuators
Environment

Utility-based Taxi Agent

Uses optimizing planning

- searches for the plan that leads to the maximum utility

There are still issues

- irreducible preference orderings
- non-deterministic environment (→ Markov decision processes)

Deductive Reasoning Agents Architecture

Symbolic Reasoning Agents

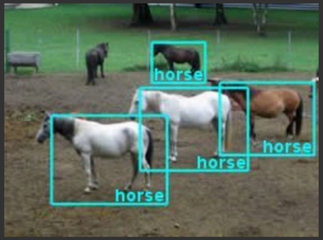
- The classical approach to building agents is to view them as a particular type of knowledge-based system, and bring all the associated methodologies of such systems to bear.
  - This paradigm is known as symbolic AI.

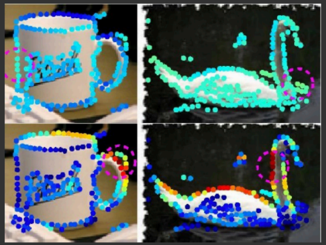
- We define a deliberative agent or agent architecture to be one that:
  - contains an explicitly represented, symbolic model of the world;
  - makes decisions (for example about what actions to perform) via symbolic reasoning.

Two challenges

The Transduction Problem

Identifying objects is hard!!!



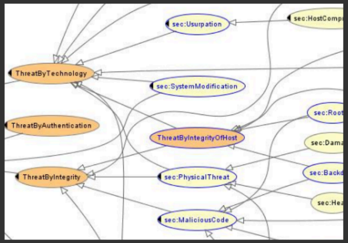


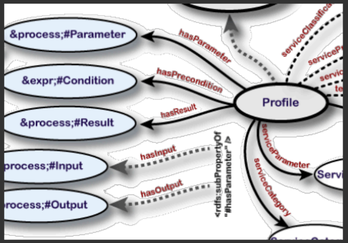
The transduction problem is that of translating the real world into an accurate, adequate symbolic description, in time for that description to be useful.

This has led onto research into vision, speech understanding, learning…

The Representation/Reasoning Problem

Representing objects is harder!





How to symbolically represent information about complex real-world entities and processes, and how to get agents to reason with this information in time for the results to be useful.

This has led onto research into knowledge representation, automated reasoning, planning…

The representation / reasoning problem

- The underlying problem with knowledge representation/ reasoning lies with the complexity of symbol manipulation algorithms.
  - In general many (most) search-based symbol manipulation algorithms of interest are ***highly intractable***.
  - Hard to find ***compact representations***.
- Because of these problems, some researchers have looked to alternative techniques for building agents; we look at these later.

Deductive Reasoning Agents

- How can an agent decide what to do using theorem proving?
  - Basic idea is to use logic to encode a theory stating the best action to perform in any given situation.

- Let:
  - ρ be this theory (typically a set of rules);
  - Δ be a logical database that describes the current state of the world;
  - Ac be the set of actions the agent can perform;
  - $\Delta \vdash_{\rho} \varphi$ means that φ can be proved from Δ using ρ.

Deductive Reasoning Agents

- How does this fit into the abstract description we talked about last time?
  - The perception function is as before:

$$see : E \to Per$$

  - of course, this is (much) easier said than done.

- The next state function revises the database Δ :

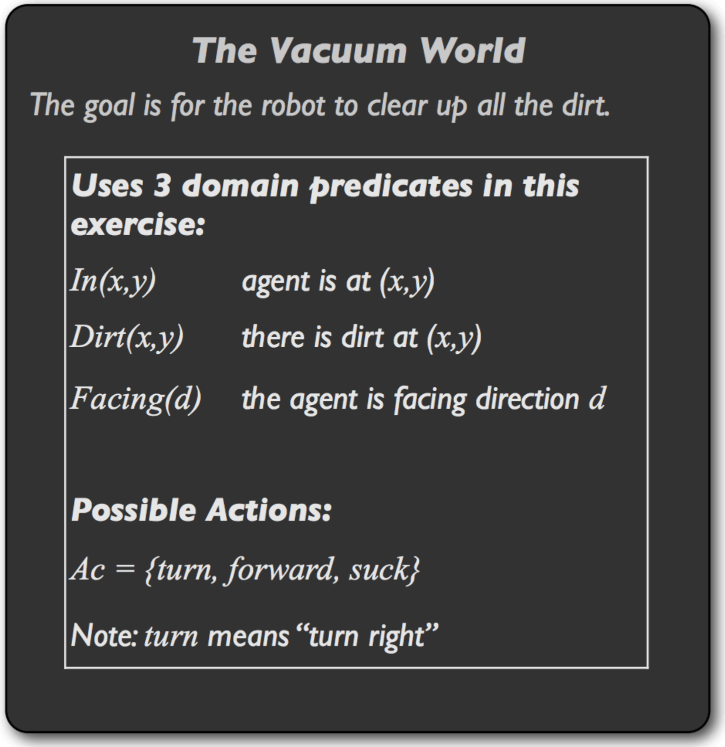
$$next : \Delta \times Per \to \Delta$$

- And the action function?
  - Well a possible action function is on the next slide.

Action Function

```
for each $\alpha \in Ac$ do            /* try to find an action explicitly prescribed */
    if $\Delta \vdash_\rho Do(\alpha)$ then
        return $\alpha$
    end-if
end-for

for each $\alpha \in Ac$ do            /* try to find an action not excluded */
    if $\Delta \nvdash_\rho \neg Do(\alpha)$ then
        return $\alpha$
    end-if
end-for

return null                            /* no action found */
```

An example: The Vacuum World



The Vacuum World

The goal is for the robot to clear up all the dirt.

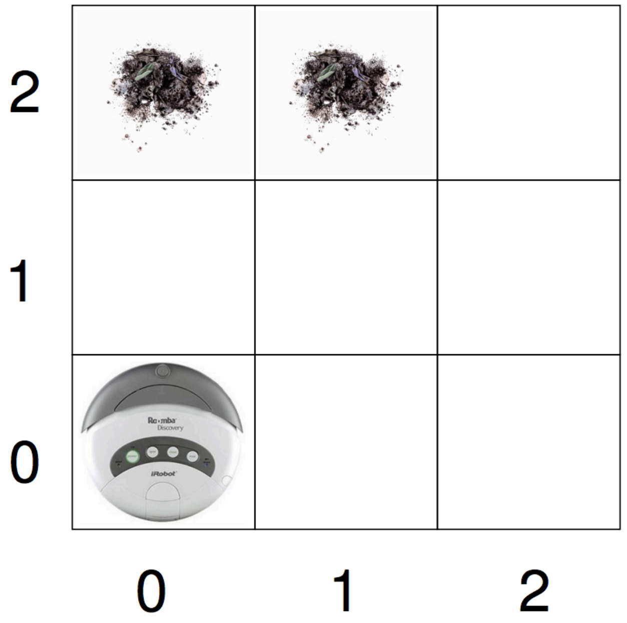
Uses 3 domain predicates in this exercise:

$In(x,y)$	agent is at (x,y)
$Dirt(x,y)$	there is dirt at (x,y)
$Facing(d)$	the agent is facing direction d

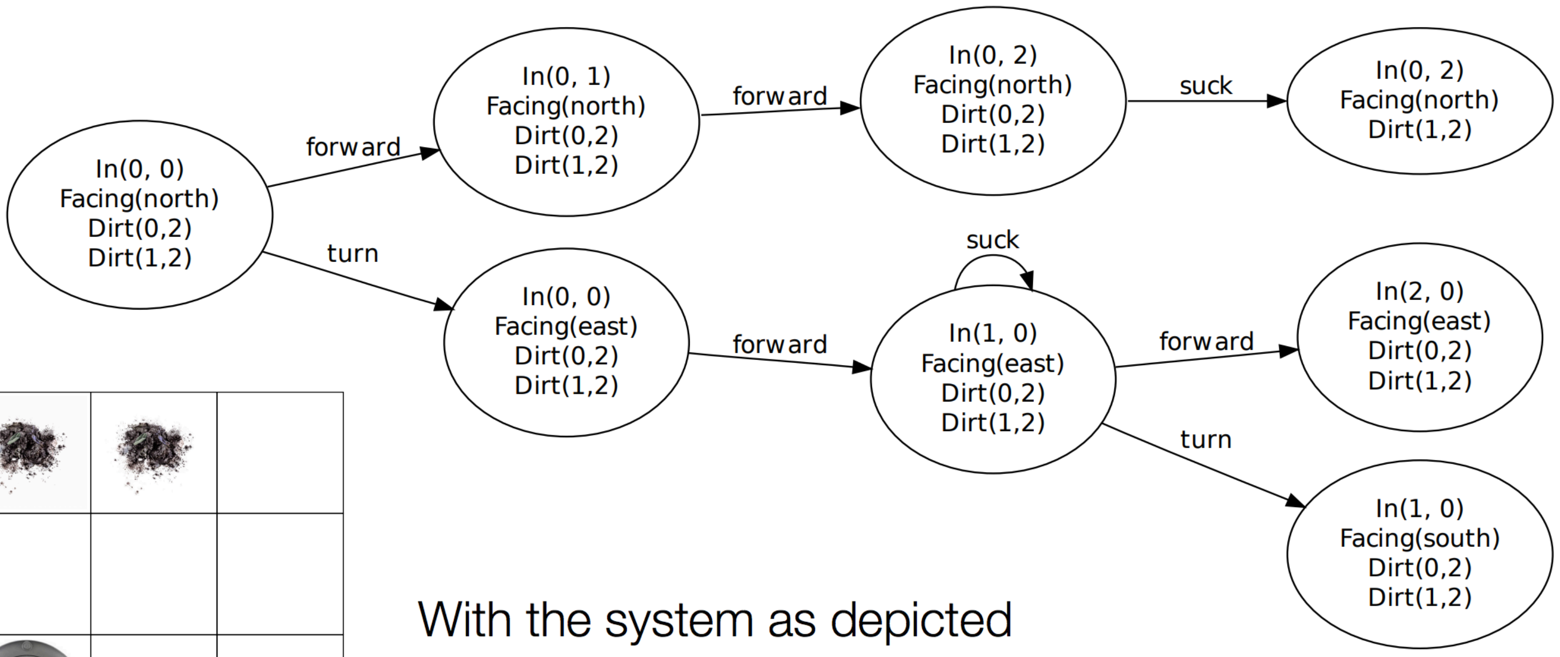
Possible Actions:

$Ac = \{turn, forward, suck\}$

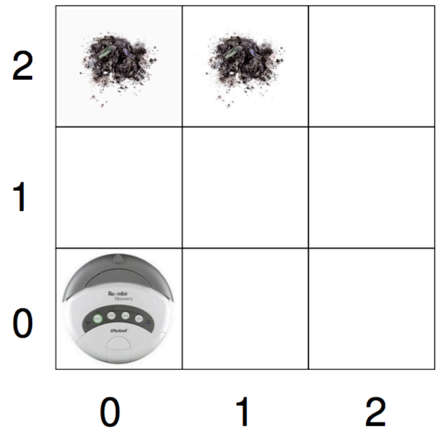
Note: *turn* means "turn right"



An example: The Vacuum World



In(0, 0) Facing(north) Dirt(0,2) Dirt(1,2)
forward
In(0, 1) Facing(north) Dirt(0,2) Dirt(1,2)
forward
In(0, 2) Facing(north) Dirt(0,2) Dirt(1,2)
suck
In(0, 2) Facing(north) Dirt(1,2)
turn
In(0, 0) Facing(east) Dirt(0,2) Dirt(1,2)
forward
suck
In(1, 0) Facing(east) Dirt(0,2) Dirt(1,2)
forward
In(2, 0) Facing(east) Dirt(0,2) Dirt(1,2)
turn
In(1, 0) Facing(south) Dirt(0,2) Dirt(1,2)



With the system as depicted above, here are some possible ways that the system might run.

An example: The Vacuum World

- Rules ρ for determining what to do:

$$
\begin{array}{rcl}
In(0,0) \wedge Facing(north) \wedge \neg Dirt(0,0) & \longrightarrow & Do(forward) \\
In(0,1) \wedge Facing(north) \wedge \neg Dirt(0,1) & \longrightarrow & Do(forward) \\
In(0,2) \wedge Facing(north) \wedge \neg Dirt(0,2) & \longrightarrow & Do(turn) \\
In(0,2) \wedge Facing(east) & \longrightarrow & Do(forward)
\end{array}
$$

  - ... and so on!

- Using these rules (+ other obvious ones), starting at *(0, 0)* the robot will clear up dirt.

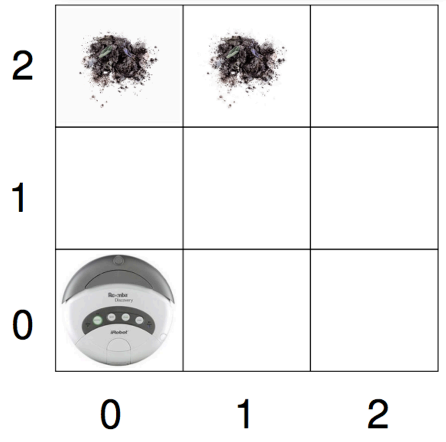
Uses 3 domain predicates in this exercise:

In(x,y)	agent is at *(x,y)*
Dirt(x,y)	there is dirt at *(x,y)*
Facing(d)	the agent is facing direction *d*

Possible Actions:

Ac = {turn, forward, suck}

Note: *turn* means "turn right"



An example: The Vacuum World

- Problems:
  - how to convert video camera input to *Dirt(0, 1)*?
  - decision making assumes a static environment:
    - ***calculative rationality***.
  - decision making using first-order logic is ***undecidable***!

- Typical solutions:
  - weaken the logic;
  - use symbolic, non-logical representations;
  - shift the emphasis of reasoning from ***run time*** to ***design time***.

Agent-oriented programming

- Yoav Shoham introduced “agent-oriented programming” in 1990:

“... new programming paradigm, based on a societal view of computation ...”

- The key idea:
  - directly programming agents in terms of intentional notions
    - like belief, desire, and intention
    - Adopts the same abstraction as humans
  - Resulted in the Agent0 programming language

AGENT0

- AGENT0 is implemented as an extension to LISP.
  - Each agent in AGENT0 has 4 components:
    - a set of ***capabilities*** (things the agent can do);
    - a set of ***initial beliefs***;
    - a set of ***initial commitments*** (things the agent will do); and
    - a set of ***commitment rules***.
- The key component, which determines how the agent acts, is the commitment rule set.
  - Each commitment rule contains
    - a ***message condition***;
    - a ***mental condition***; and
    - an ***action***.

AGENT0 Decision Cycle

On each decision cycle . . .

- The message condition is matched against the messages the agent has received;
  - The mental condition is matched against the beliefs of the agent.
  - If the rule fires, then the agent becomes committed to the action (the action gets added to the agents commitment set).

Actions may be . . .

- Private
  - An externally executed computation
- Communicative
  - Sending messages

Messages are constrained to be one of three types . . .

- requests
  - To commit to action
- unrequests
  - To refrain from action
- Informs
  - Which pass on information

AGENT0 Commitment Rules

- This rule may be paraphrased as follows:
  - if I receive a message from *agent* which requests me to do *action* at *time*, and I believe that:
    - *agent* is currently a friend;
    - I can do the *action*;
    - at *time*, I am not committed to doing any other *action*,
  - then commit to doing *action* at *time*.

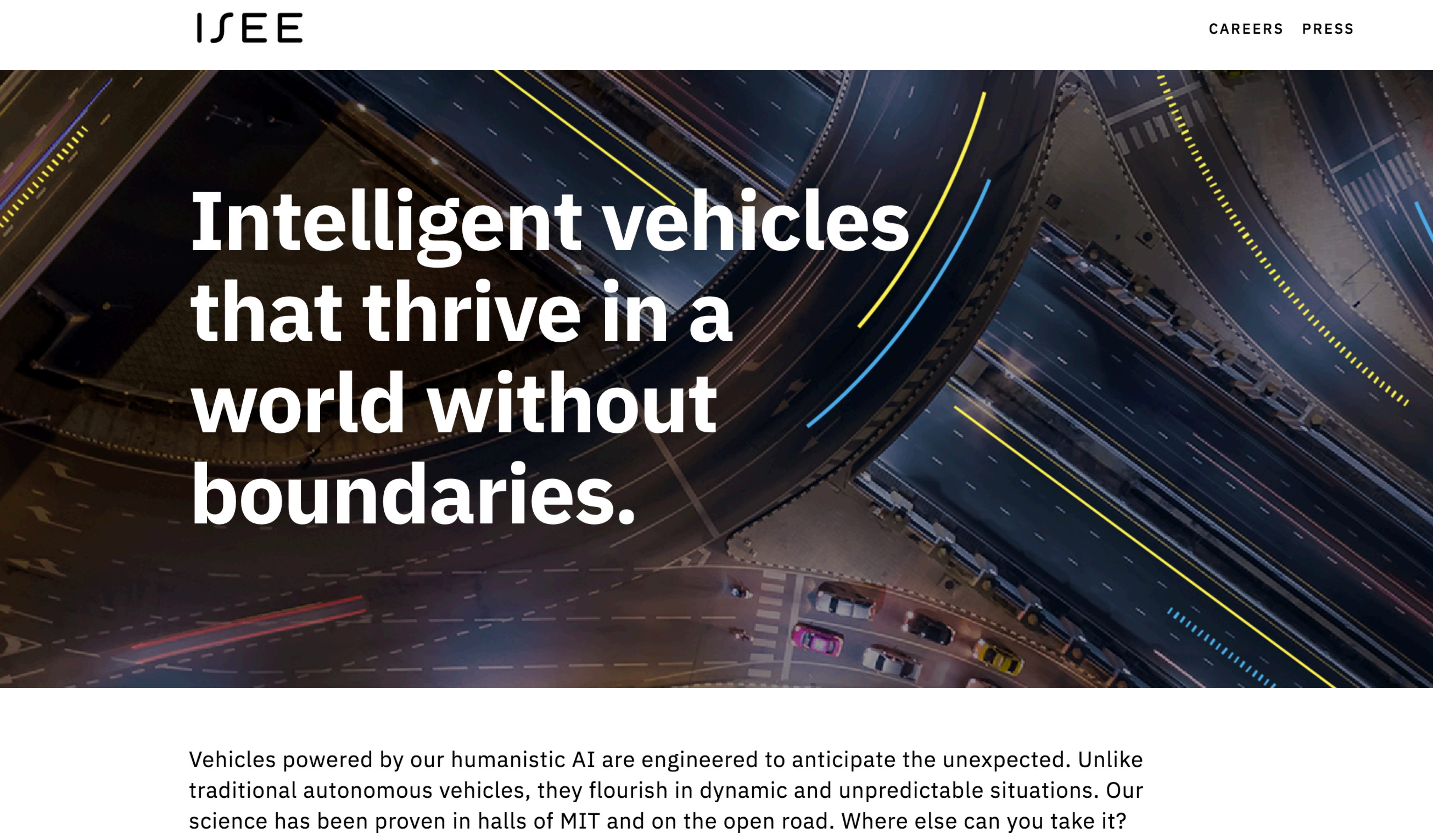
A commitment Rule

```
COMMIT(
  ( agent, REQUEST, DO(time, action)
  ), ;;; msg condition
  ( B,
    [now, Friend agent] AND
    CAN(self, action) AND
    NOT [time, CMT(self, anyaction)]
  ), ;;; mental condition
  self,
  DO(time, action)
)
```

AGENT Programming Languages

1990: AGENT-0 (Shoham)	Speech acts
1993: PLACA (Thomas; AGENT-0 extension with plans)	Plans
1996: AgentSpeak(L) (Rao; inspired by PRS)	Events/Intentions
1996: Golog (Reiter, Levesque, Lesperance)	Action theories, logical specification
1997: 3APL (Hindriks et al.)	Practical reasoning rules
1998: ConGolog (Giacomo, Levesque, Lesperance)	Capabilities, Java-based
2000: JACK (Busetta, Howden, Ronnquist, Hodgson)	Declarative goals
2000: GOAL (Hindriks et al.)	Mobile agents (within agent commu
2000: CLAIM (Amal El FallahSeghrouchni)	AgentSpeak + Communication
2002: Jason (Bordini, Hubner; implementation of AgentSpeak)	JADE + BDI
2003: Jadex (Braubach, Pokahr, Lamersdorf)	Modules, PG-rules, …
2008: 2APL (successor of 3APL)	

BDI Agent Architecture



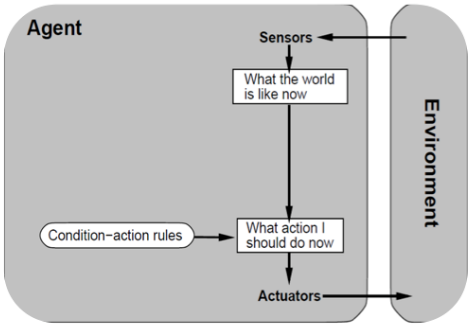
ISEE

CAREERS PRESS

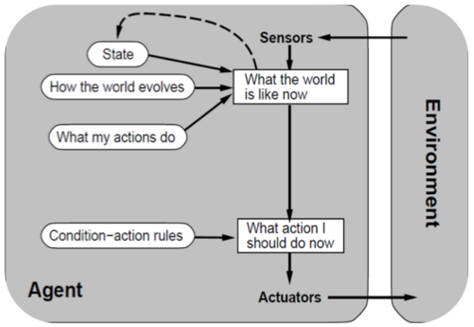
Intelligent vehicles that thrive in a world without boundaries.

Vehicles powered by our humanistic AI are engineered to anticipate the unexpected. Unlike traditional autonomous vehicles, they flourish in dynamic and unpredictable situations. Our science has been proven in halls of MIT and on the open road. Where else can you take it?

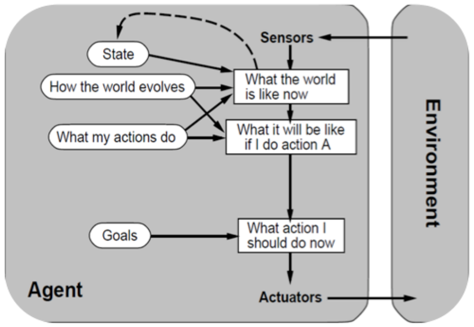
Basic Agent Architectures



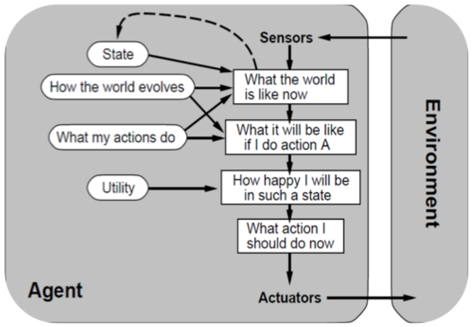
Reflex agent



Model-based agent

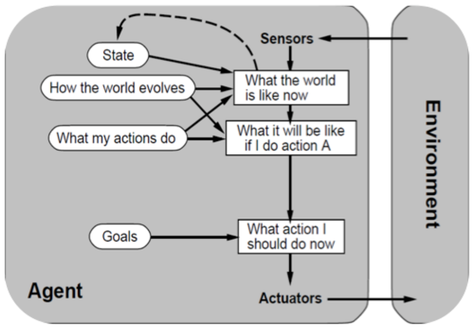

Sensors
State
How the world evolves
What my actions do
What the world is like now
What it will be like if I do action A
Goals
What action I should do now
Agent
Actuators
Environment

Goal-based agent


Sensors
State
How the world evolves
What my actions do
What the world is like now
What it will be like if I do action A
Utility
How happy I will be in such a state
What action I should do now
Agent
Actuators
Environment

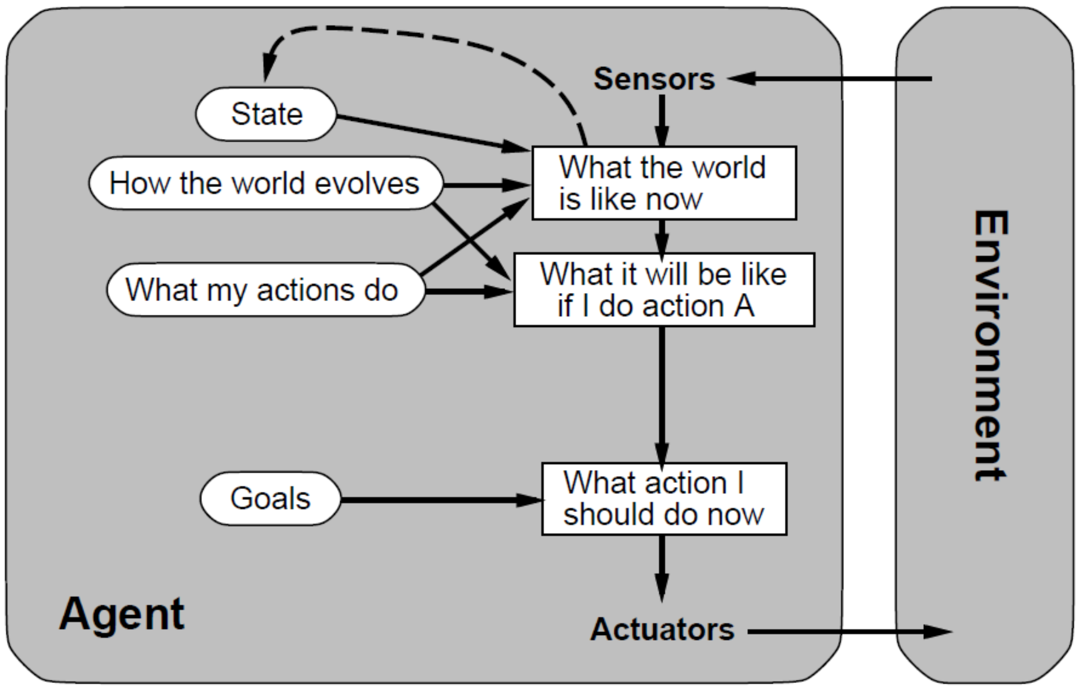
Utility-based agent

Basic Agent Architectures


Sensors
State
What the world is like now
How the world evolves
What my actions do
What it will be like if I do action A
Goals
What action I should do now
Agent
Actuators
Environment

Goal-based agent

Goal-based agents


State
How the world evolves
What my actions do
Goals
Sensors
What the world is like now
What it will be like if I do action A
What action I should do now
Actuators
Agent
Environment

How to go from goals to actions effectively?

Big Picture

philosophical foundations		Practical reasoning		Deductive reasoning
analysis and design	BDI architecture	other architecture		Various (modal) logics
implementation	Agent programming languages	Interpreters / Execution architectures	solvers, theorem provers	Interpreters / Execution architectures

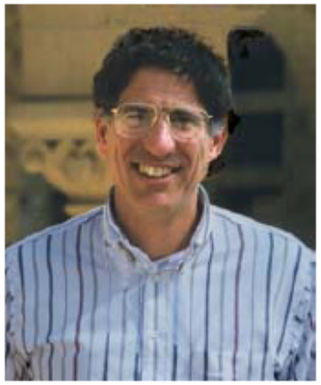
Practical Reasoning

Conceptualizing rational action

Practical Reasoning

- Practical reasoning is reasoning directed towards actions — the process of figuring out what to do.
- Principles of practical reasoning applied to agents largely derive from work of philosopher Michael Bratman (1990):

  "Practical reasoning is a matter of weighing conflicting considerations for and against competing options, where the relevant considerations are provided by what the agent desires/values/cares about and what the agent believes."



- Distinguish practical reasoning from theoretical reasoning.

Theoretical vs Practical Reasoning

"In theory, there is no difference between theory and practice. But, in practice, there is." – Jan L. A. van de Snepscheut

1. Theoretical reasoning is reasoning directed towards beliefs — concerned with deciding what to believe.
   - Tries to assess the way things are.
   - Process by which you change your beliefs and expectations;.
   - Example: you believe q if you believe p and you believe that if p then q.
2. Practical reasoning is reasoning directed towards actions — concerned with deciding what to do.
   - Decides how the world should be and what individuals should do.
   - Process by which you change your choices, plans, and intentions.
   - Example: you go to class, if you must go to class.

The Components of Practical Reasoning

Human practical reasoning consists of two activities:

strategic

1. Deliberation: deciding what state of affairs we want to achieve.
   - considering preferences, choosing goals, etc.;
   - balancing alternatives (decision-theory);
   - the outputs of deliberation are intentions;
   - interface between deliberation and means-end reasoning.

tactical

2. Means-ends reasoning: deciding how to achieve these states of affairs:
   - thinking about suitable actions, resources and how to "organize" activity;
   - building courses of action (planning);
   - the outputs of means-ends reasoning are plans.

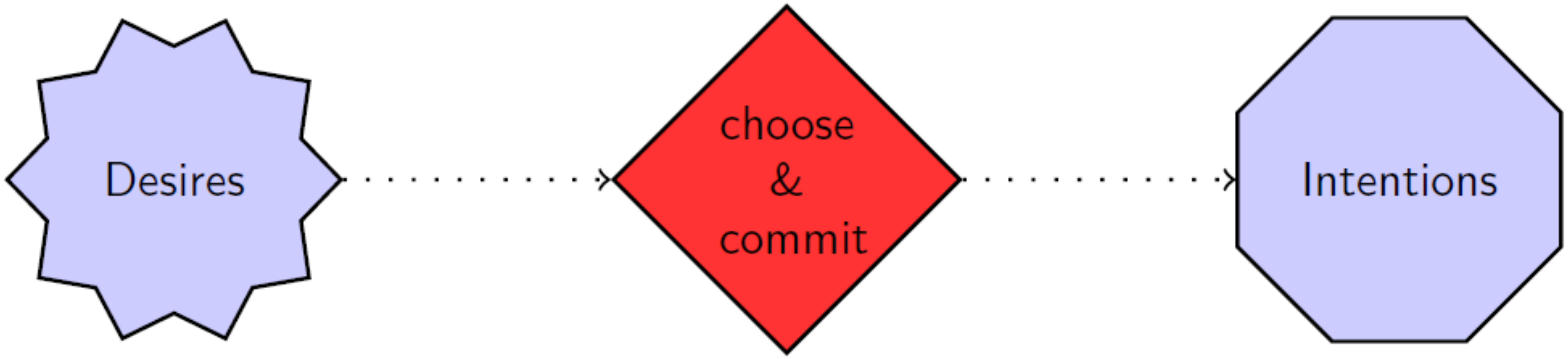
Fact: agents are resource-bounded & world is dynamic!

The key: To combine **deliberation** & **means-ends reasoning** appropriately.

Deliberation

How does an agent deliberate?

1. Begin by trying to understand what the options available to you are:
   - options available are desires.
2. Choose between them, and commit to some:
   - chosen options are then intentions.



Desires

- Desires describe the states of affairs that are considered for achievement, i.e., basic preferences of the agent.
- Desires are much weaker than intentions; not directly related to activity:

> *"My desire to play basketball this afternoon is merely a potential influence of my conduct this afternoon. It must vie with my other relevant desires [...] before it is settled what I will do. In contrast, once I intend to play basketball this afternoon, the matter is settled: I normally need not continue to weigh the pros and cons. When the afternoon arrives, I will normally just proceed to execute my intentions."*
> *(Bratman 1990)*

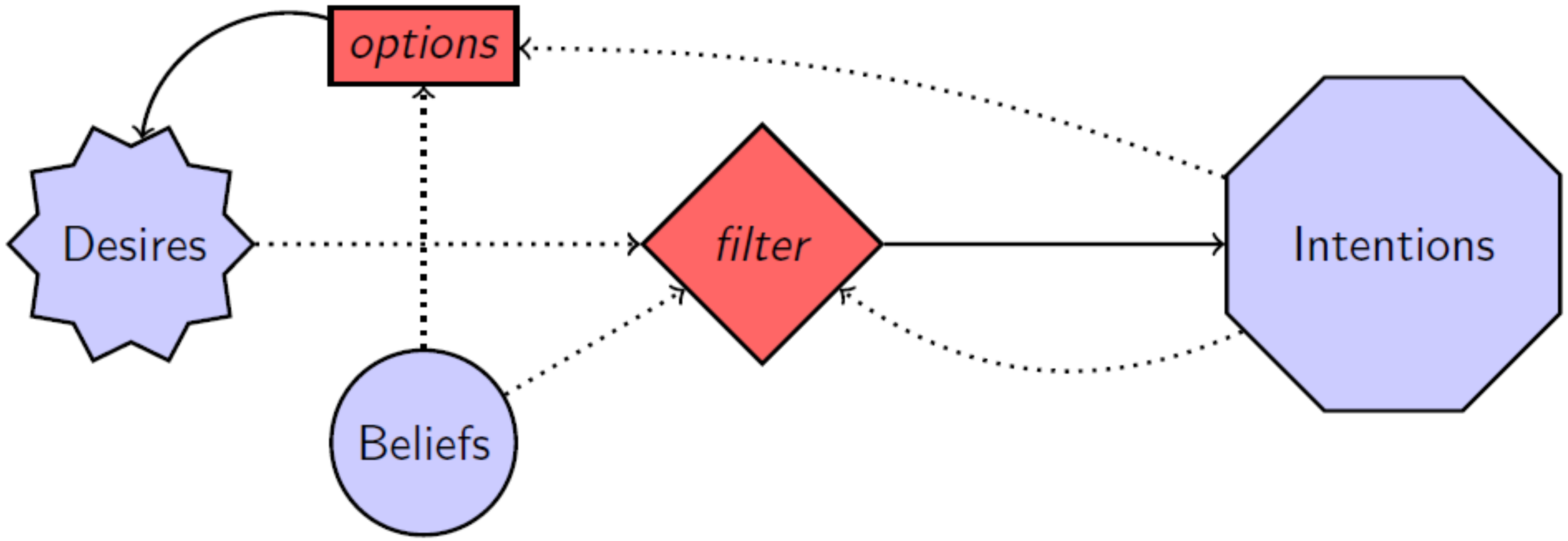
Intentions

- In ordinary speech: intentions refer to actions or to states of mind;
  - here we consider the latter!
  - E.g., I may adopt/have the intention to be an academic.
- Focus on future-directed intentions i.e. pro-attitudes leading to actions.
  - Intentions are about the (desired) future.
- We make reasonable attempts to fulfill intentions once we form them, but they may change if circumstances do.
  - Behavior arises to fulfill intentions.
  - Intentions affect action choice.

Functional Components of Deliberation

Option Generation agent generates a set of possible alternatives; via a function, *options*, which takes the agent's current beliefs and intentions, and from them determines a set of options/desires.

Filtering in which the agent chooses between competing alternatives, and commits to achieving them. In order to select between competing options, an agent uses a filter function.



Properties of Intentions

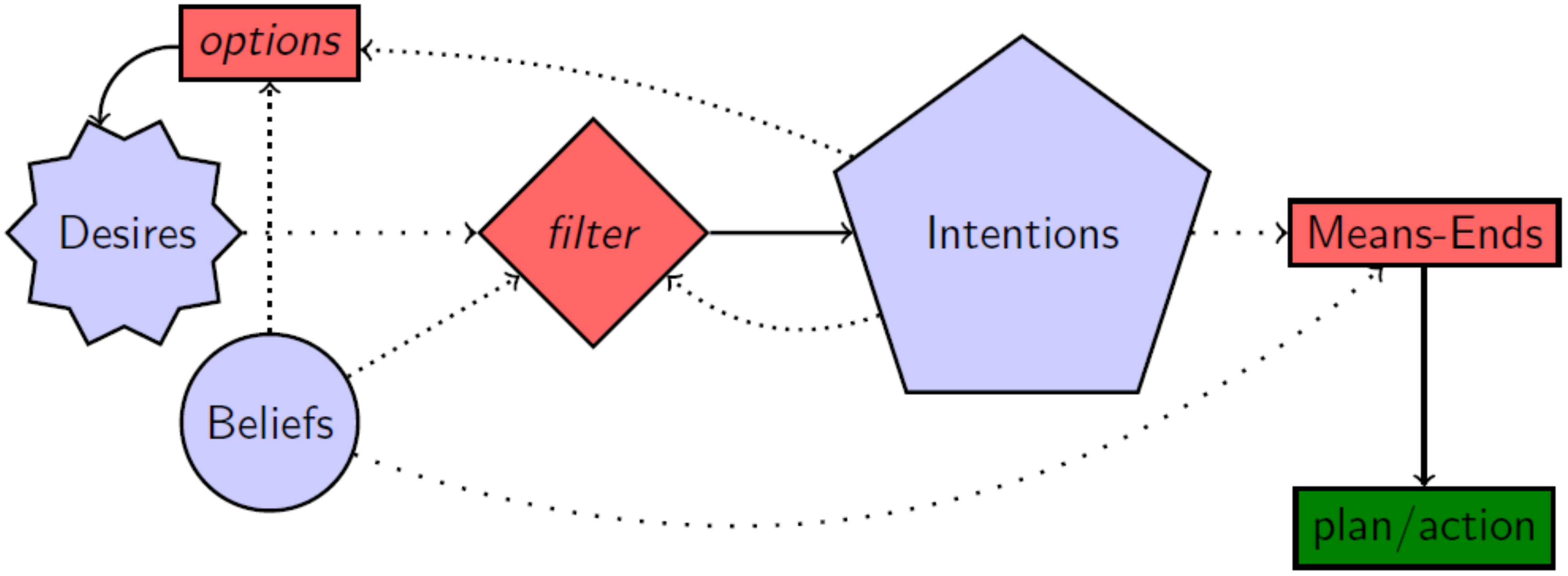
1. Intentions drive means-end reasoning.
2. Intentions constrain future deliberation (i.e., provide a “filter”).
3. Intentions persist.
4. Intentions influence beliefs concerning future practical reasoning.
5. Agents believe their intentions are possible.
6. Agents do not believe they will not bring about their intentions.
7. Under certain circumstances, agents believe they will bring about their intentions.
8. Agents need not intend all the expected side effects of their intentions.

Plans

Human practical reasoning consists of two activities:

1. Deliberation: deciding what to do. Forms intentions.
2. Means-ends reasoning: deciding how to do it. Forms plans. Forms plans.

Intentions drive means-ends reasoning: *If I adopt an intention, I will attempt to achieve it, this affects action choice.*



Commitments

We may think that deliberation and planning are sufficient to achieve desired behavior, unfortunately things are more complex...

After filter function, agent makes a commitment to chosen option:

- Commitment: *an agreement or pledge to do something in the future*;
- $\therefore$ it implies temporal persistence.

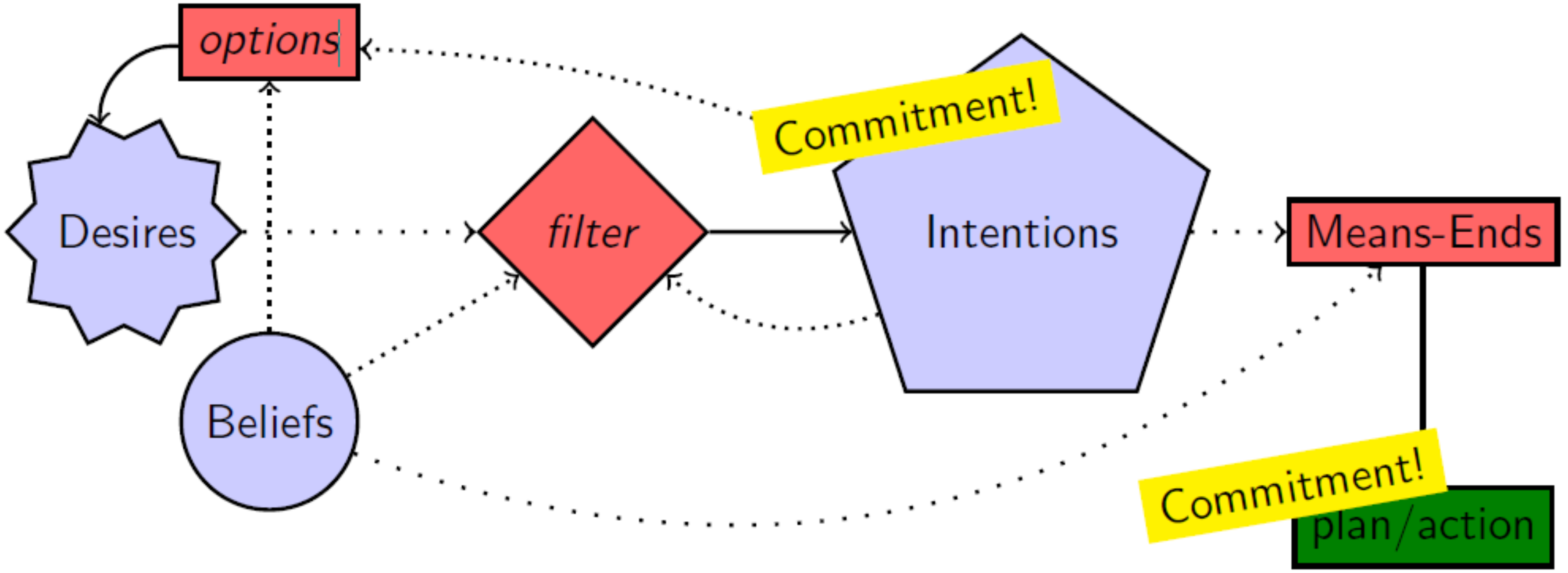
Questions:

1. how long should an intention persist?
2. what is the commitment on?

Commitments to Ends and Means

An agent has commitment both to ends (intentions), and means (plans).

- I am committed to meet/see my friend John this week (an intention);
- I am committed to drop-by John's place on Thursday afternoon (a mean).



Degrees of Commitments

Rao and Georgeff (1991) described the following commitment strategies:

Blind/Fanatical commitment A blindly committed agent will continue to maintain an intention until it believes the intention has actually been achieved.

Single-minded commitment A single-minded agent will continue to maintain an intention until it believes that either the intention has been achieved, or else that it is no longer possible to achieve the intention.

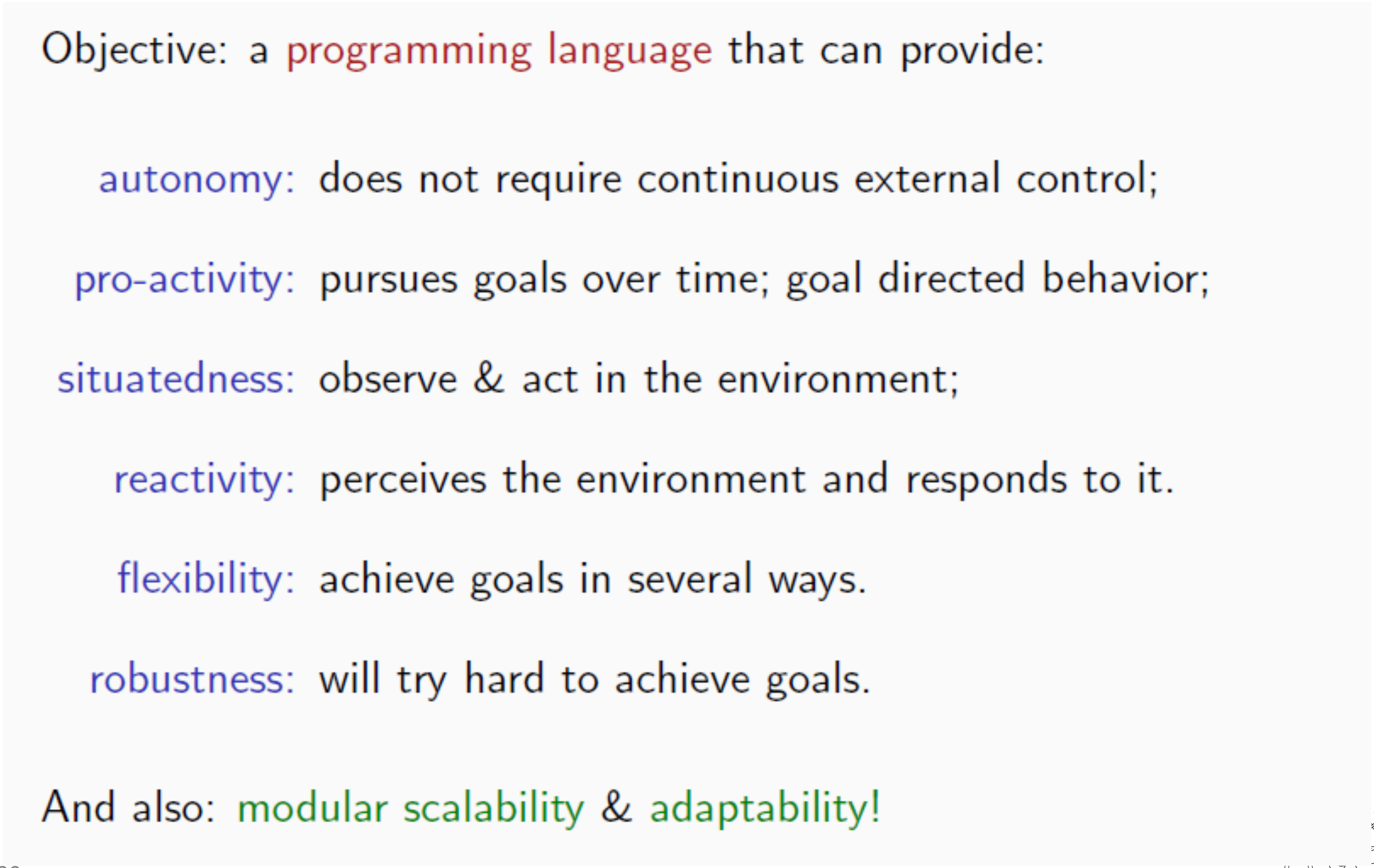
Open-minded commitment An open-minded agent will maintain an intention until until achieved as long as it is still believed possible.

BDI Programming

Operationalizing practical reasoning

ti.

What is BDI Programming Language?



Objective: a programming language that can provide:

autonomy:	does not require continuous external control;
pro-activity:	pursues goals over time; goal directed behavior;
situatedness:	observe & act in the environment;
reactivity:	perceives the environment and responds to it.
flexibility:	achieve goals in several ways.
robustness:	will try hard to achieve goals.

And also: modular scalability & adaptability!

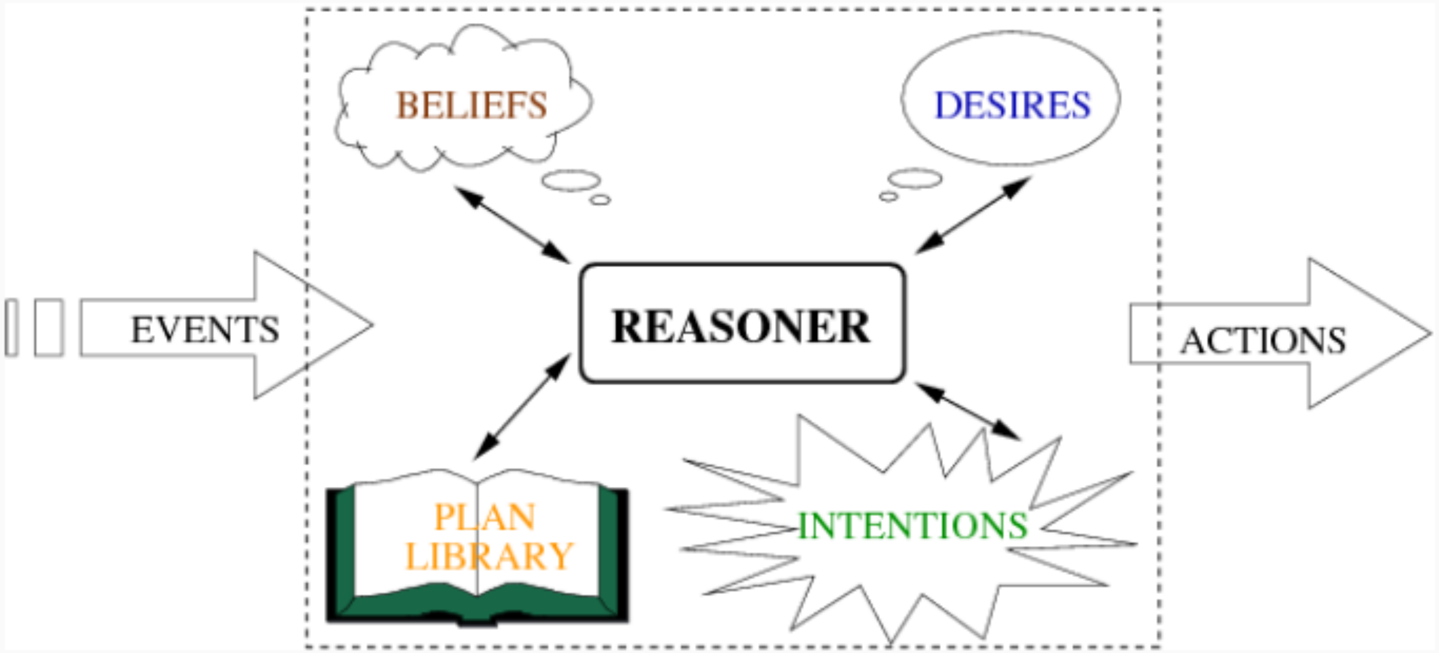
What is BDI Programming Language?

We want to program intelligent systems under the following constraints:

1. The agent interacts with an external environment.
   - A grid world with gold pieces, obstacles, and other agents.
2. The environment is (highly) dynamic; may change in unexpected ways.
   - Gold pieces appear randomly.
3. Things can go wrong; plans and strategies may fail.
   - A path may end up being blocked.
4. Agents have dynamic and multiple objectives.
   - Explore, collect, be safe, communicate, etc.
   - Motivations/goals/desires may come and go.

What is BDI Programming Language?

Beliefs:	information about the world.
Events:	goals/desires to resolve; internal or external.
Plan library:	recipes for handling goals-events.
Intentions:	partially uninstantiated programs with commitment.



Intentions

1. Agent's intentions are determined dynamically by the agent at runtime based on its known facts, current goals, and available plans.

2. An intention is just a partially executed strategy:
   - comes from the plan library when resolving events.

3. An intention represent a focus of attention:
   - something the agent is currently working on;
   - actions/behavior arises as a consequence of executing intentions.

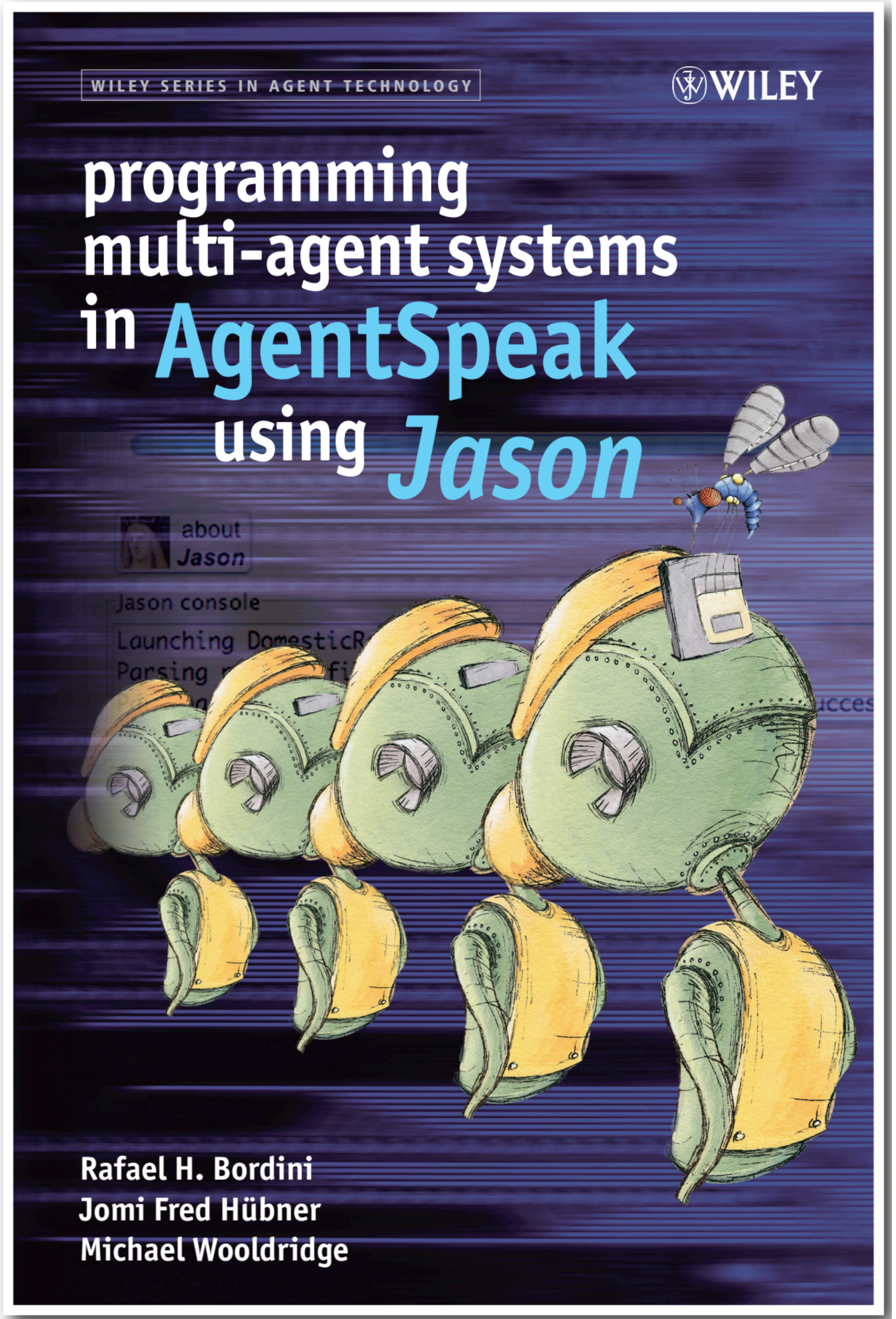
4. An agent may have several intentions active at one time.
   - different simultaneous focuses of attention;

5. A new intention is created when an external event is addressed.

6. An intention may create/post an internal event:
   - the intention will be updated when this event is addressed.

AgentSpeak (L)

- Developed by A. S. Rao and has been influential in the design of other agent programming languages.
- Programming language for implementing BDI architectures.
- Extended to make it a practical agent programming language (R. Bordini).
- AgentSpeak programs can be executed by the Jason interpreter (R. Bordini et al.).
  - http://jason.sourceforge.net/
- Based on logic programming (Prolog) using restricted first-order language with events and actions.
  - There are also non-logic-based agent programming languages.


WILEY SERIES IN AGENT TECHNOLOGY
WILEY
programming multi-agent systems in AgentSpeak using Jason
about Jason
Jason console
Rafael H. Bordini
Jomi Fred Hübner
Michael Wooldridge

AgentSpeak

- The main language constructs of AgentSpeak are:
  - Beliefs
  - Goals
  - Plans
- The architecture of an AgentSpeak agent has four main components:
  - Belief Base
  - Plan Library
  - Set of Events
  - Set of Intentions

AgentSpeak - Beliefs and Goals

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

```
publisher(wiley)
```

- Goals represent states of affairs the agent wants to bring about (come to believe, when goals are used declaratively)

  - Achievement goals:

```
!write(book)
```

Or attempts to retrieve information from the belief base

  - Test goals:

```
?publisher(P)
```

AgentSpeak -- Events and Plans

- An agent reacts to events by executing plans
- Events happen as a consequence to changes in the agent's beliefs or goals
- Plans are recipes for action, representing the agent's know-how
- An AgentSpeak plan has the following general structure:

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

- where:
  - the triggering event denotes the events that the plan is meant to handle;
  - the context represent the circumstances in which the plan can be used;
  - 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.

AgentSpeak -- Events and Plans

- AgentSpeak triggering events:
  - `+b` (belief addition)
  - `-b` (belief deletion)
  - `+!g` (achievement-goal addition)
  - `-!g` (achievement-goal deletion)
  - `+?g` (test-goal addition)
  - `-?g` (test-goal deletion)
- The context is logical expression, typically a conjunction of literals to be checked whether they follow from the current state of the belief base
- The body is a sequence of actions and (sub) goals to achieve.
- NB: This is the original AgentSpeak syntax; ***Jason*** allows other things in the context and body of plans.

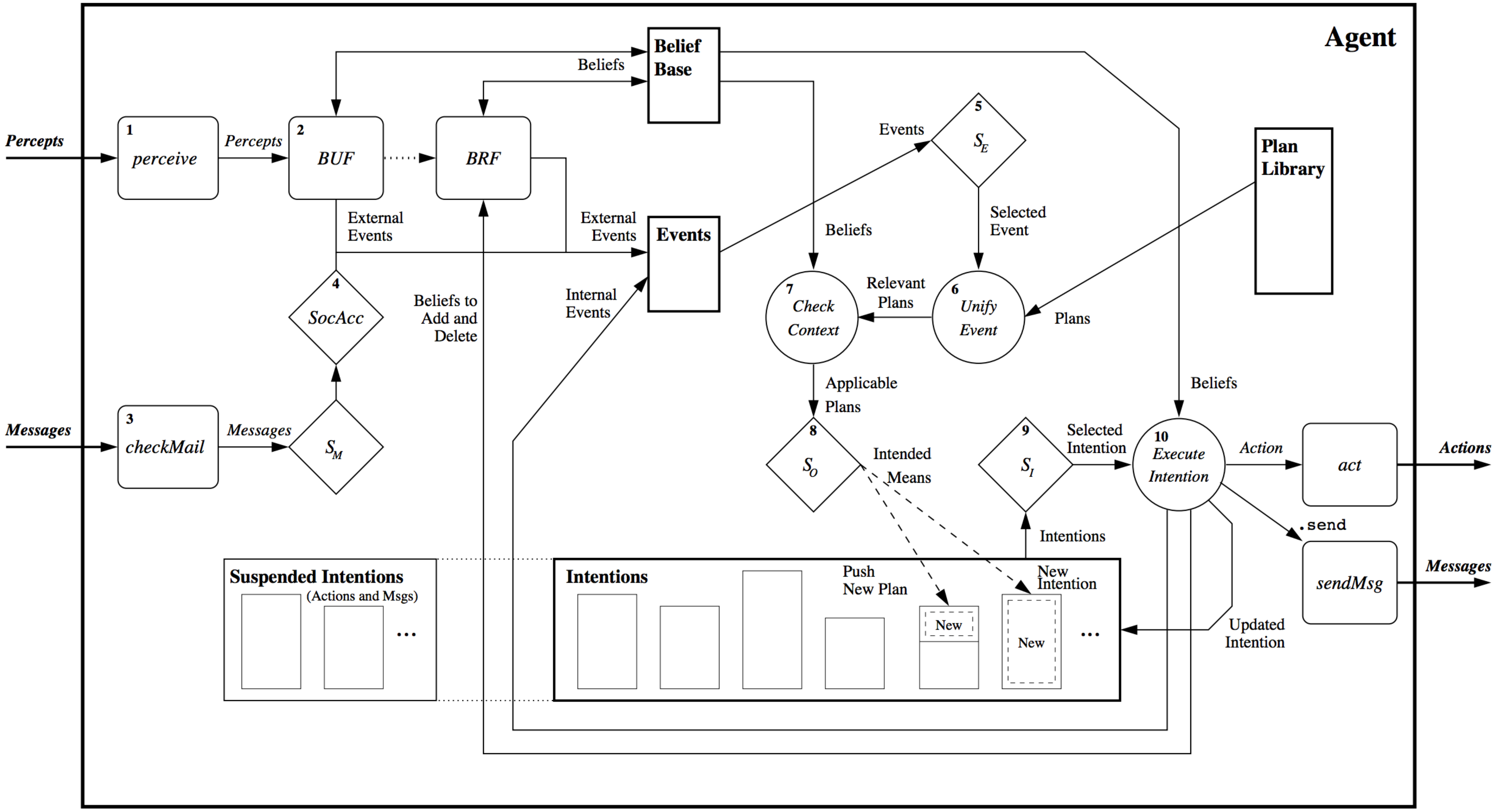
AgentSpeak -- Events and Plans

```
+green_patch(Rock)
   :  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) ...
```

Jason

- ***Jason*** implements the operational semantics of a variant of AgentSpeak
- Various extensions aimed at a more practical programming language
- Platform for developing multi-agent systems
- Developed by Jomi F. Hübner and Rafael H. Bordini
- We'll look at the ***Jason*** additions to AgentSpeak and its features

Jason Reasoning Cycle



AgentSpeak: Example

ALICE

- During lunch time, forward all calls to Carla.
- When I am busy, incoming calls from colleagues should be

AgentSpeak Example Plans

```
user(alice).
user(bob).
user(carla).
user(denise).
~status(alice, idle).
status(bob, idle).
colleague(bob).
lunch_time(‘‘11:30’’).
```

AgentSpeak Example Plans

```
user(alice).
user(bob).
user(carla).
user(denise).
~status(alice, idle).
status(bob, idle).
colleague(bob).
lunch_time(“11:30”).

+invite(X, alice) :
    lunch_time(t)   ←   !call_forward(alice, X, carla).                (p1)
+invite(X, alice) :
    colleague(X)  ← call_forward_busy(alice,X,denise).                 (p2)
+invite(X, Y):
    true   ←  connect(X,Y).                                            (p3)

+!call_forward(X, From, To) :
    invite(From, X) ← +invite(From, To), - invite(From,X)              (p4)

+!call_forward_busy(Y, From, To) :
    invite(From, Y)& not(status(Y, idle)))
    ← +invite(From, To), - invite(From,Y).                             (p5)
```

Execution - 1

A new event is sensed from the environment, `+invite(Bob, Alice)` (there is a call for Alice from Bob).

There are three relevant plans for this event (p1, p2 and p3)

- the event matches the triggering event of those three plans.

Relevant Plans	**Unifier**

`p1: +invite(X, alice) : lunch_time(t)`
 `←!call_forward(alice, X, carla)`	

`p2: +invite(X, alice) : colleague(Bob)`
 `← !call_forward_busy(alice, X, denise).`
	{X=bob}

`p3 : +invite(X, Y): true ← connect(X,Y).`	{Y=alice, X=bob}

of plan p2 is satisfied - c
=> p2 is applicable.

Execution - 2

Context of plan p2 is satisfied - `colleague(bob)` => p2 is *applicable*.

A new intention based on this plan is created in the set of intentions, because the event was external, generated from the perception of the environment.

The plan starts to be executed. It adds a new event, this time an internal event: `!call_forward_busy(alice,bob,denise).`

Intention ID	Intention Stack	Unifier

1	`+invite(X,alice):colleague(X)`
	`<- !call_forward_busy(alice,X,denise)`
		{X=bob}

Execution - 3

A plan relevant to this new event is found (p5):

Relevant Plans	**Unifier**

`p5: +!call_forward_busy(Y, From, To) :`
`invite(From, Y) & not(status(Y, idle)))`
`← +invite(From, To),`
`- invite(From,Y).`
	{From=bob,
	Y=alice,
	To=denise}

p5 has the context condition true, so it becomes an *applicable* plan and it is pushed on top of *intention 1 (*it was generated by an internal event)

Intention ID	**Intention Stack**	**Unifier**

2	`+!call_forward_busy(Y,From,To) :`
	`invite(From,Y) & not status(Y,idle)`
	`<- +invite(From,To); -invite(From,Y)`
		{From=bob,
		Y=alice,
		To=denise}

1	`+invite(X,alice) : colleague(X)`
	`<- !call_forward_busy(alice,X,denise)`
		{X=bob}

Execution - 4

A new internal event is created, **`+invite(bob, denise).`**

three relevant plans for this event are found, p1, p2 and p3.

However, only plan p3 is applicable in this case, since the others don't have the context condition true.

The plan is pushed on top of the existing intention.

Intention ID	**Intension Stack**	**Unifier**
3	`+invite(X,Y) : <- connect(X,Y)`	{Y=denise, X=bob}

2	`+!call_forward_busy(Y,From,To) :`
	`invite(From,Y) & not status(Y,idle)`
	`<- +invite(From,To); -invite(From,Y)`
		{From=bob, Y=alice, To=denise}

1	`+invite(X,alice) : colleague(X)`
	`<- !call_forward_busy(alice,X,denise)`
		{X=bob}

Execution - 5

On top of the intention is a plan whose body contains an action.

The action is executed, **`connect(bob, denise)`** and is removed from the intention.

When all formulas in the body of a plan have been removed (i.e., have been executed), the whole plan is removed from the intention, and so is the achievement goal that generated it.

Intention ID	**Intension Stack**	**Unifier**

1	`+!call_forward_busy(Y,From,To) :`
	`invite(From,Y) & not status(Y,idle)`
	`<- -invite(From,Y)`
		{From=bob,
		Y=alice,
		To=denise}

	`+invite(X,alice) : colleague(X)`
	`<- !call_forward_busy(alice,X,denise)`
		{X=bob}

- The only thing that remains to be done is **`-invite(bob, alice)`** (this event is removed from the beliefs base).
- This ends a cycle of execution, and the process starts all over again, checking the state of the environment and reacting to events.

Jason x Prolog

- With the ***Jason*** extensions, nice separation of theoretical and practical reasoning
- BDI arcthicture allows
  - long-term goals (goal-based behaviour)
  - reacting to changes in a dynamic environment
  - handling multiple foci of attention (concurrency)
- Acting on an environment and a higher-level conception of a distributed system
- Direct integration with Java