Multiagent Systems I

Prof. Dr. Jürgen Dix

Department of Informatics
Clausthal University of Technology
SS 2010

Time: Monday, Tuesday: 10–12
Place: T2 (lecture), Ifl R301 (labs)
Labs: From 27. April on.

Website

http://www.in.tu-clausthal.de/abteilungen/cig/cigroot/teaching

Visit regularly!

Lecture: Prof. Dix, T. Behrens, M. Köster
Labs: T. Behrens, M. Köster

Schein: Lab work

About this Lecture

This course gives a first introduction to multi-agent systems for Bachelor students. Emphasis is put on applications and programming MAS, not on theory. We consider one programming language together with a platform for developing agents: JASON. Students are grouped into teams and implement agent teams for solving a task on our agent contest platform. These teams compete against each other.

My thanks go to Tristan Behrens, Michael Köster and our students who prepared the lab work and also some of the slides of this course. In addition, Mehdi Dastani and Jomi Hübner provided me with some slides.

References


Wooldridge, M. (2002).
An Introduction to Multi Agent Systems.
John Wiley & Sons.


Rafael H. Bordini, Jomi Fred Hübner and Michael Wooldridge (2007).
Programming Multi-Agent Systems in AgentSpeak using Jason.
John Wiley & Sons.

Lecture Overview

1. Week: **1. Introduction**
2. Week: **2. Jason**
3. Week: **3. Jason Logic Programming**
4.-14. Week: **Labs.**

Outline

1. Introduction

Content of this Chapter:

We are setting the stage for a precise discussion of agency. From **informal concepts** to (more or less) mathematical definitions.

1. MAS versus **Distributed AI (DAI)**,
2. Environment of agents,
3. Agents and other frameworks,
4. Runs as characteristic behaviour,
5. state-based versus standard agents.

1.1 Why Agents?

Three Important Questions

(Q1) What is a (software) agent?

(Q2) If some program P is not an agent, how can it be transformed into an agent?

(Q3) If (Q1) is clear, what kind of Software Infrastructure is needed for the interaction of agents? What services are necessary?

Definition 1.1 (Distributed AI (DAI))

The area investigating systems, where several autonomous acting entities work together to reach a given goal.

The entities are called *Agents*, the area Multiagent Systems.

AAMAS: several conferences joined in 2002 to form the main annual event. Bologna (2002), Melbourne (2003), New York (2004), Utrecht (2005), Hakodate (2006), Hawaii (2007), Lisbon (2008), Budapest (2009).

Example 1.2 (RoboCup)

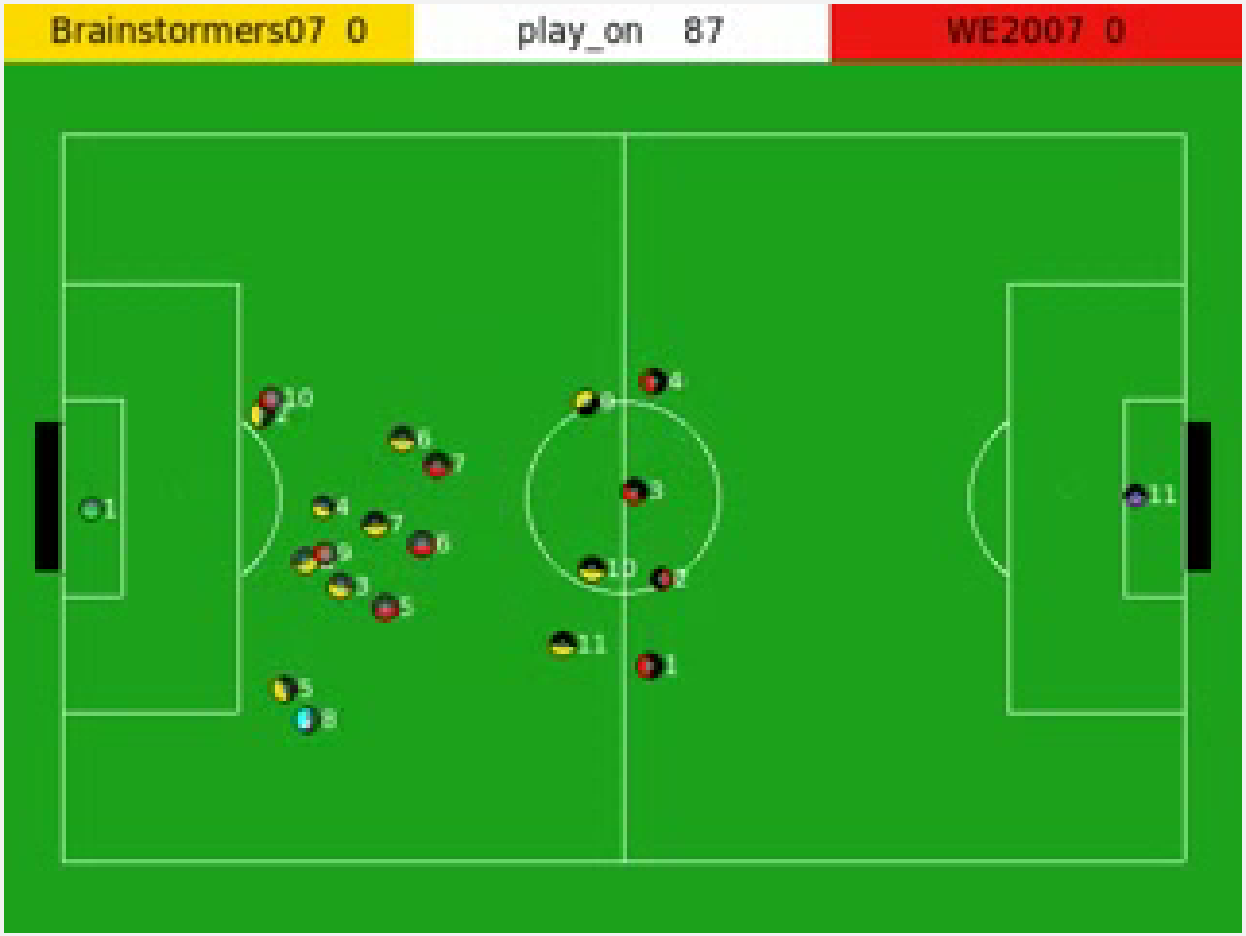


Figure : 2D-Simulation league: RoboCup 2007 Final

Example 1.3 (RoboCup)

Figure : 3D-Simulation league: RoboCup 2007 Final

Example 1.4 (RoboCup)

Figure : Small size league

Example 1.5 (RoboCup)

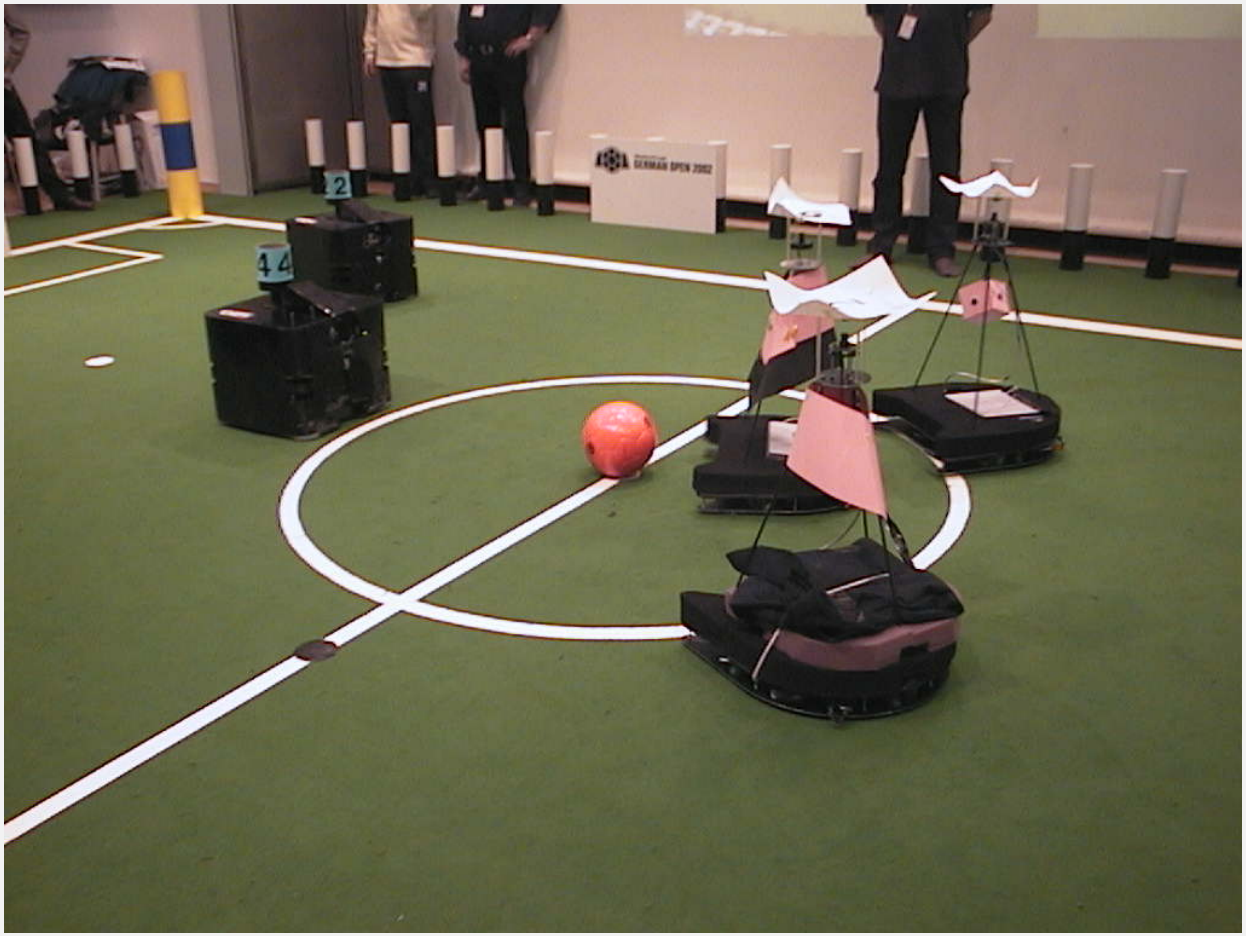


Figure : Middle size league

Example 1.6 (RoboCup)

Figure : Standard platform

Example 1.7 (RoboCup)

Figure : Humanoid league

Example 1.8 (RoboCup)

Figure : Rescue league

Example 1.9 (Grand Challenge 2004)

Grand Challenge: Organised by DARPA since 2004.
First try: **Huge Failure**.

Figure : Grand Challenge 2004

- Prize money: 1 million Dollars
- Race course: 241 km in the Mojave desert
- 10 hours pure driving time
- More than 100 registered participants, 15 of them were chosen
- **No one reached the end of the course**
- **The favourite** "Sandstorm" of Carnegie Mellon in Pittsburgh **managed 5%** of the distance

Example 1.10 (Grand Challenge 2005)

Second try: **Big Success**:
Stanley (Sebastian Thrun) won in 2005.

Figure : VW Touareg coached by Stanford University

- Prize money: 2 million Dollars
- Race course: 212,76 km in the Mojave desert
- 10 hours pure driving time
- 195 registered participants, 23 were qualified
- **5 teams** reached the end of the course (**4 teams in time**)
- **Stanley** finished the race in 6 hours and 53 minutes (30,7 km/h)
- **Sandstorm** achieved the second place

Example 1.11 (Urban Challenge)

Urban Challenge: Organised by DARPA since 2007.

Figure : Urban Challenge 2007

- No straight-line course but real streets covered with buildings.
- 60 miles
- Prize money: 3,5 million Dollars
- **Tartan Racing** won, **Stanford Racing Team** second, **VictorTango** third place.
- Some teams like **Stanford Racing Team** and **VictorTango** as well as **Tartan Racing** were sponsored by **DARPA** with 1 million Dollar beforehand.

Example 1.12 (CLIMA Contest: Gold Mining)

First try: A simple grid where agents are supposed to collect gold. Different roles of agents: scouts, collectors.

- `http://multiagentcontest.org`

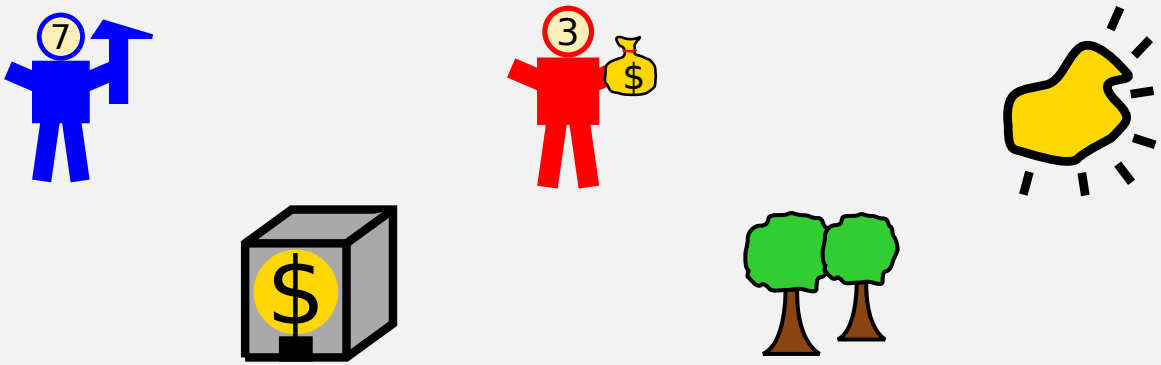


Figure : Gold mining elements

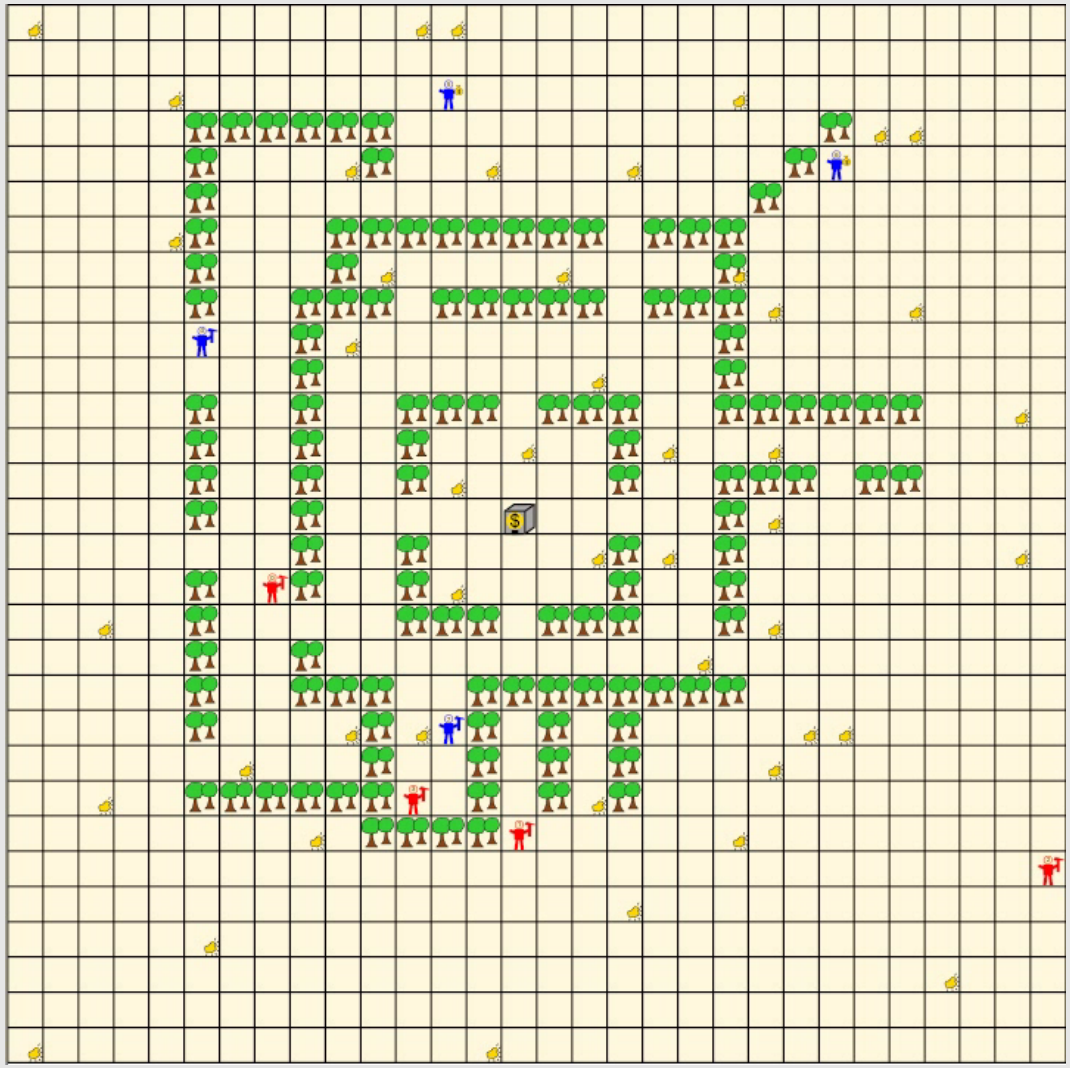


Figure : Gold Mining 2006: CLIMABot (blue) vs. brazil (red)

Example 1.13 (Agent Contest: Chasing Cows)

Second try: Push cows in a corral.

- http://multiagentcontest.org

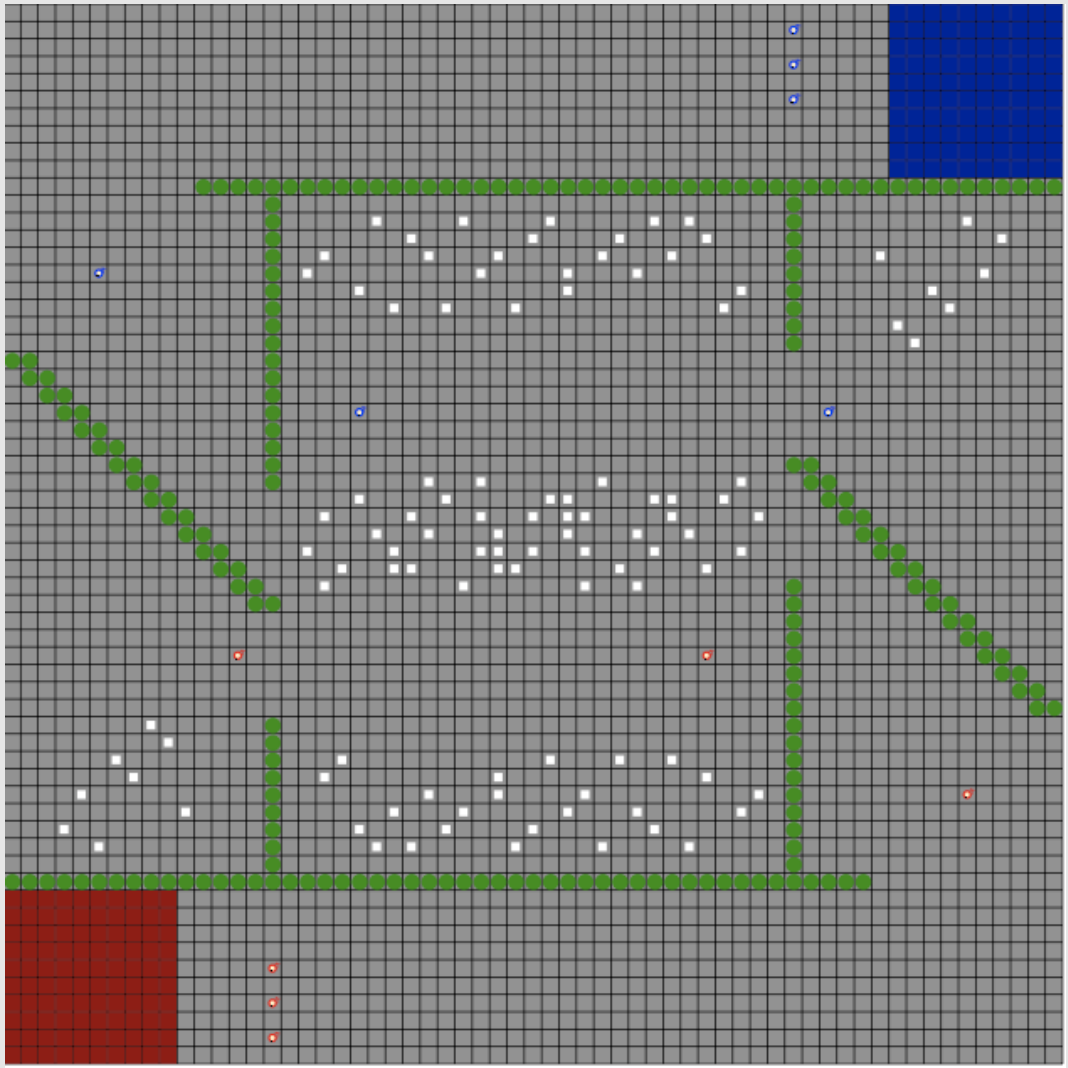


Figure : Chasing Cows 2008

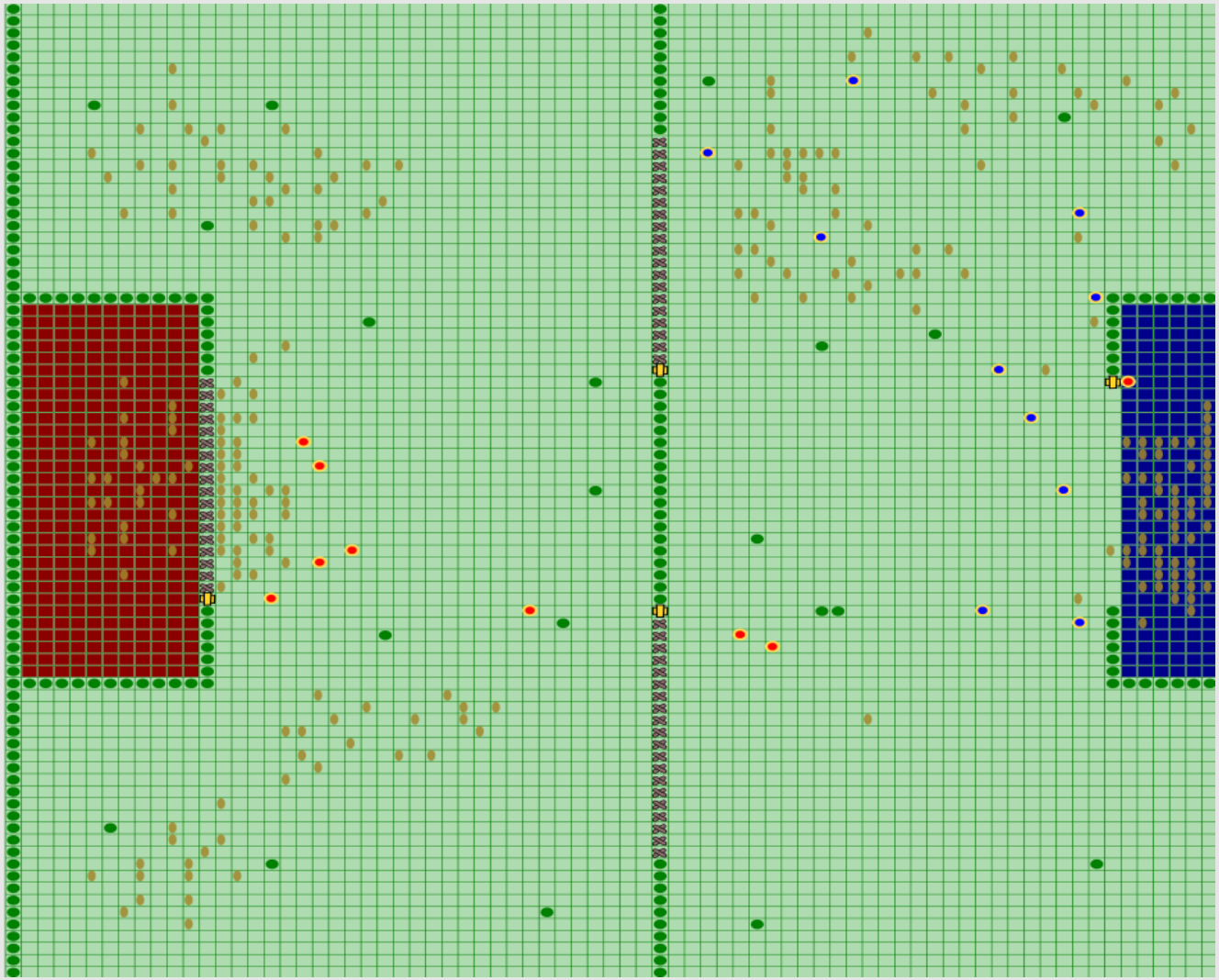


Figure : Chasing Cows 2009

Agents: Why do we need them?

Information systems are **distributed**, **open**, **heterogenous**.
We therefore need intelligent, interactive agents, that act autonomously.

(Software) Agent: Programs that are implemented on a platform and have **sensors** and **effectors** to read from and make changes to the environment, respectively.

Intelligent: Performance measures, to evaluate the success. **Rational** vs. **omniscient**, **decision making**

Interactive: with other agents (software or humans) by observing the environment.
Coordination: **Cooperation** vs. **Competition**

MAS versus Classical DAI

MAS: Several Agents coordinate their knowledge and actions (semantics describes this).

DAI: Particular problem is divided into smaller problems (nodes). These nodes have common knowledge. **The solution method is given.**

Attention:

Today DAI is used synonymously with MAS.

AI	DAI
Agent	**Multiple** Agents
Intelligence: Property of a **single** Agent	Intelligence: Property of several Agents
Cognitive Processes of a **single** Agent	Social Processes of several Agents

10 Desiderata

1. Agents are for everyone! We need a method to agentise given programs.
2. Take into account that **data is stored in a wide variety of data structures, and data is manipulated by an existing corpus of algorithms.**
3. A theory of agents must *not* depend upon the set of actions that the agent performs. Rather, **the set of actions that the agent performs must be a *parameter* that is taken into account in the semantics.**

10 Desiderata

4. **Every (software) agent should execute actions based on some *clearly articulated* decision policy.** A declarative framework for articulating decision policies of agents is imperative.
5. Any agent construction framework must allow agents to reason:
   - **Reasoning about its beliefs** about other agents.
   - **Reasoning about uncertainty** in its beliefs about the world and about its beliefs about other agents.
   - **Reasoning about time**.

These capabilities should be viewed as *extensions* to a core agent action language.

10 Desiderata

6. **Any infrastructure to support multiagent interactions *must* provide security.**
7. While the efficiency of the code underlying a software agent cannot be guaranteed (as it will vary from one application to another), **guarantees are needed that provide information on the performance of an agent relative to an oracle that supports calls to underlying software code.**

10 Desiderata

8. We must identify efficiently computable fragments of the general hierarchy of languages alluded to above, and our implementations must take advantage of the specific structure of such language fragments.
9. A critical point is reliability—there is no point in a highly efficient implementation, if all agents deployed in the implementation come to a grinding halt when the agent "infrastructure" crashes.

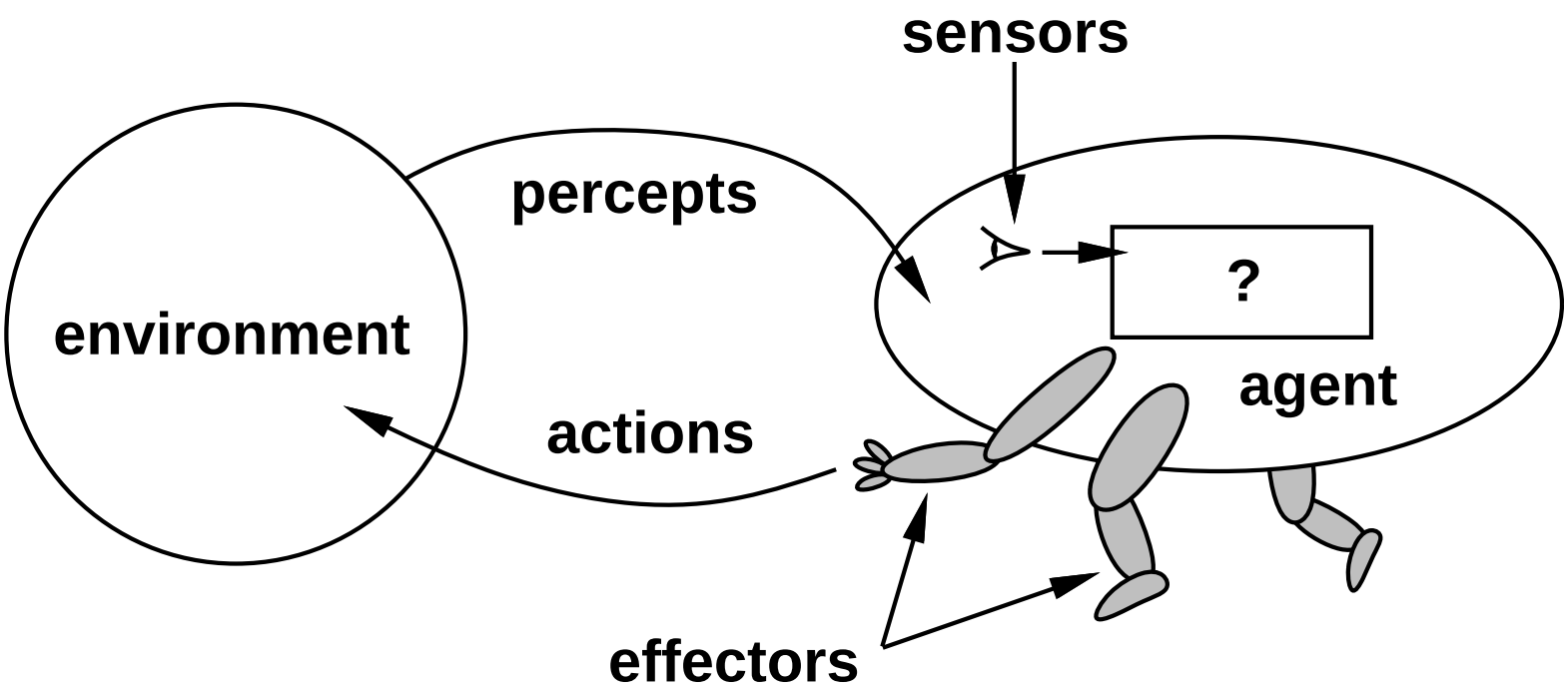
10 Desiderata

10. The only way of testing the applicability of any theory is to build a software system based on the theory, to deploy a set of applications based on the theory, and to report on experiments based on those applications.

1.2 Intelligent Agents

Definition 1.14 (Agent a)

An agent a is anything that can be viewed as **perceiving** its environment through **sensor** and **acting** upon that environment through **effectors**.



Definition 1.15 (Rational, Omniscient Agent)

A rational agent is one that does the right thing (**Performance measure** determines how successful an agent is).

A omniscient agent knows the actual outcome of his actions and can act accordingly.

Attention:

A rational agent is in general not omniscient!

Question

What is the right thing and what does it depend on?

1. Performance measure (as objective as possible).
2. Percept sequence (everything the agent has received so far).
3. The agent's knowledge about the environment.
4. How the agent can act.

Definition 1.16 (Ideal Rational Agent)

For each possible percept-sequence an ideal rational agent should do whatever action is expected to maximize its performance measure (based on the evidence provided by the percepts and built-in knowledge).

Mappings:

$$\text{set of percept sequences} \mapsto \text{set of actions}$$

can be used to describe agents in a mathematical way.

Hint:

Internally an agent is

agent = architecture + program

AI is engaged in designing agent programs

Agent Type	Perform. Measure	Environment	Actuators	Sensors
Medical diagnosis system	Healthy patient, minimize costs	Patient, hospital, staff	Display questions, tests, diagnoses, treatments	Entry of symptoms, findings, patient's answers
Satellite image analysis system	Correct image categorization	Downlink from orbiting satellite	Display categorization of scene	Color pixel arrays
Part-picking robot	Percentage of parts in correct bins	Conveyor belt with parts; bins	Jointed arm and hand	Camera, joint angle sensors
Interactive English tutor	Maximize student's score on test	Set of students, testing agency	Display exercises, suggestions, corrections	Keyboard entry

Table : Examples of agents types and their **PEAS** descriptions.

Question:

How do environment properties influence agent design?

Definition 1.17 (Environment Properties)

Accessible/Inaccessible: If not completely accessible, one needs internal states.

Deterministic/Indeterministic: An inaccessible environment might seem indeterministic, even if it is not.

Episodic/Nonepisodic: Percept-Action-Sequences are independent from each other. Closed episodes.

Static/Dynamic: While the agent is thinking, the world is the same/changing. **Semi-dynamic**: The world does not change, but the performance measure.

Discrete/Continous: Density of observations and actions. Relevant: Level of granularity.

Environment	Accessible	Deterministic	Episodic	Static	Discrete
Chess with a clock	Yes	Yes	No	Semi	Yes
Chess without a clock	Yes	Yes	No	Yes	Yes
Poker	No	No	No	Yes	Yes
Backgammon	Yes	No	No	Yes	Yes
Taxi driving	No	No	No	No	No
Medical diagnosis system	No	No	No	No	No
Image-analysis system	Yes	Yes	Yes	Semi	No
Part-picking robot	No	No	Yes	No	No
Refinery controller	No	No	No	No	No
Interactive English tutor	No	No	No	No	Yes

xbiff, software demons are agents (not intelligent).

Definition 1.18 (Intelligent Agent)

An intelligent agent is an agent with the following properties:

1. Autonomous: Operates without direct intervention of others, has some kind of control over its actions and internal state.
2. Reactive: Reaction to changes in the environment at certain times to reach its goals.
3. Pro-active: Taking the initiative, being goal-directed.
4. Social: Interaction with others to reach the goals.

Pro-active alone is not sufficient (C-Programs): The environment can change during execution.

Socialisation: coordination, communication, (negotiation) skills.

Difficulty: right balance between pro-active and reactive!

Agents vs. Object Orientation I

Objects have

1. a state (encapsulated): control over internal state
2. message passing capabilities

Java: private and public methods.

- Objects have control over their state, but **not over their behaviour**.
- An object can not prevent others to use its public methods.

Agents vs. Object Orientation II

Agents call other agents and request them to execute actions.

- Objects do it for free, agents do it for money.
- No analoga to **reactive**, **pro-active**, **social** in OO.
- MAS are multi-threaded or even multi-processed: each agent has a control thread or is a new process. (In OO only the system as a whole possesses one.)

A Simple Agent Program

function SKELETON-AGENT(*percept*) **returns** action
static: *memory*, the agent's memory of the world

memory ← UPDATE-MEMORY(*memory*, *percept*)
action ← CHOOSE-BEST-ACTION(*memory*)
memory ← UPDATE-MEMORY(*memory*, *action*)
return *action*

In Theory Everything is Trivial

function TABLE-DRIVEN-AGENT(*percept*) **returns** *action*
  static: *percepts*, a sequence, initially empty
    table, a table, indexed by percept sequences, initially fully specified

  append *percept* to the end of *percepts*
  action ← LOOKUP(*percepts, table*)
  return *action*

Example 1.19 (Agent: Taxi Driver)

PEAS description of the task environment for an automated taxi:

Performance Measure: Safe, fast, legal, maximize profits

Environment: Roads, other traffic, pedestrians, customers

Actuators: Steering, accelerator, brake, signal, horn

Sensors: Cameras, sonar, GPS, odometer, engine sensors

Example 1.20 (Agent: Taxi Driver)

1. **Production rules:** If the driver in front hits the breaks, then hit the breaks too.

function SIMPLE-REFLEX-AGENT(*percept*) **returns** *action*
static: *rules*, a set of condition-action rules

state ← INTERPRET-INPUT(*percept*)
rule ← RULE-MATCH(*state*, *rules*)
action ← RULE-ACTION[*rule*]
return *action*

Agents as Intentional Systems

Intentions: Agents are endowed with **mental states**.

Matthias took his umbrella because he **believed** it was going to rain.
Kjeld attended the MAS course because he **wanted** to learn about agents.

An intentional system describes entities whose behaviour can be predicted by the method of attributing beliefs, desires and rational acumen.

1.3 Formal Description

A First Mathematical Description

At first, we want to keep everything as simple as possible.

Agents and environments

An agent is situated in an environment and can perform actions

$$A := \{a_1, \ldots, a_n\} \qquad \text{(set of actions)}$$

and change the state of the environment

$$S := \{s_1, s_2, \ldots, s_n\} \qquad \text{(set of states).}$$

How does the environment (the state $\mathbf{s}$) develop when an action $\mathbf{a}$ is executed?

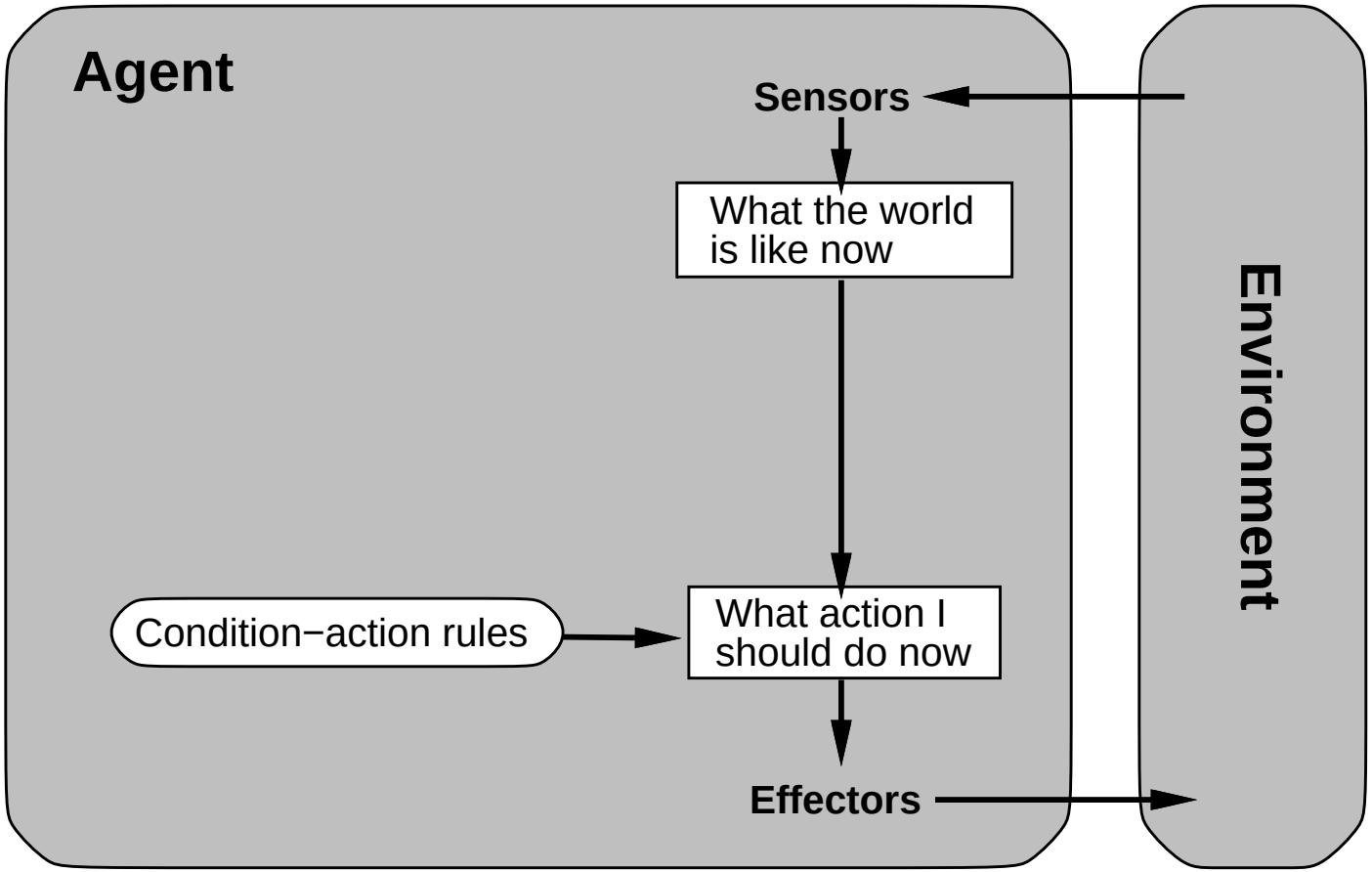
We describe this with a function

$$\mathbf{env} : \mathbf{S} \times \mathbf{A} \longrightarrow \mathbf{2^S}.$$

This includes non-deterministic environments.

How do we describe agents?

We could take a function $\mathbf{action} : \mathbf{S} \longrightarrow \mathbf{A}$.



Question:

How can we describe an agent, now?

Definition 1.21 (Purely Reactive Agent)

An agent is called purely reactive, if its function is given by

$$\mathbf{action} : \mathbf{S} \longrightarrow \mathbf{A}.$$

This is too weak!

Take the whole history (of the environment) into account: $\mathbf{s_0} \rightarrow_{\mathbf{a_0}} \mathbf{s_1} \rightarrow_{\mathbf{a_1}} \ldots \mathbf{s_n} \rightarrow_{\mathbf{a_n}} \ldots$.

The same should be done for $\mathbf{env}$!

This leads to agents that take the **whole sequence of states** into account, i.e.

$$\mathbf{action} : \mathbf{S}^* \longrightarrow \mathbf{A}.$$

We also want to consider the actions **performed by an agent**. This requires the notion of a run (next slide).

We define the run of an agent in an environment as a sequence of interleaved states and actions:

Definition 1.22 (Run r, $\mathrm{R} = \mathrm{R}^{act} \cup \mathrm{R}^{state}$)

A run r over $\mathbf{A}$ and $\mathbf{S}$ is a finite sequence

$$\mathbf{r} : \mathbf{s_0} \rightarrow_{\mathbf{a_0}} \mathbf{s_1} \rightarrow_{\mathbf{a_1}} \ldots \mathbf{s_n} \rightarrow_{\mathbf{a_n}} \ldots$$

Such a sequence may end with a state $\mathbf{s_n}$ or with an action $\mathbf{a_n}$: we denote by R^{act} the set of **runs ending with an action** and by R^{state} the set of **runs ending with a state**.

Definition 1.23 (Environment, 2. version)

An **environment** Env is a triple $\langle \mathrm{S}, \mathrm{s_0}, \tau \rangle$ consisting of

1. the set S of states,
2. the initial state $\mathrm{s_0} \in \mathrm{S}$,
3. a function $\tau : \mathrm{R}^{act} \longrightarrow 2^{\mathrm{S}}$, which describes how the environment changes when an action is performed (given the whole history).

Definition 1.24 (Agent a)

An agent a is determined by a function

$$\mathbf{action} : \mathrm{R}^{state} \longrightarrow \mathbf{A},$$

describing which action the agent performs, given its current history.

Important:

An agent system is then a pair $\mathtt{a} = \langle \mathbf{action}, Env \rangle$ consisting of an agent and an environment.
We denote by $\mathrm{R}(\mathtt{a}, Env)$ the set of runs of agent a in environment Env.

Definition 1.25 (Characteristic Behaviour)

The characteristic behaviour of an agent $\mathbf{a}$ in an environment Env is the set R of all possible runs $\mathbf{r} : \mathbf{s_0} \rightarrow_{a_0} \mathbf{s_1} \rightarrow_{a_1} \ldots \mathbf{s_n} \rightarrow_{a_n} \ldots$ with:

1. for all n: $\mathbf{a_n} = \mathbf{action}(\langle \mathbf{s_0}, \mathbf{a_0} \ldots, \mathbf{a_{n-1}}, \mathbf{s_n} \rangle)$,
2. for all $n > 0$: $\mathbf{s_n} \in \boldsymbol{\tau}(\mathbf{s_0}, a_0, \mathbf{s_1}, a_1, \ldots, \mathbf{s_{n-1}}, a_{n-1})$.

For deterministic τ, the relation "$\in$" can be replaced by "=".

Important:

The formalization of the characteristic behaviour is dependent of the concrete agent type. Later we will introduce further behaviours (and corresponding agent designs).

Equivalence

Two agents $\mathtt{a}$, $\mathtt{b}$ are called **behaviourally equivalent wrt. environment** Env, if $\mathrm{R}(\mathtt{a}, Env) = \mathrm{R}(\mathtt{b}, Env)$.
Two agents $\mathtt{a}$, $\mathtt{b}$ are called behaviourally equivalent, if they are behaviourally equivalent wrt. all possible environments Env.

So far so good, but...

What is the problem with all these agents and this framework in general?

Problem

All agents have perfect information about the environment!

(Of course, it can also be seen as feature!)

We need more realistic agents!

Note

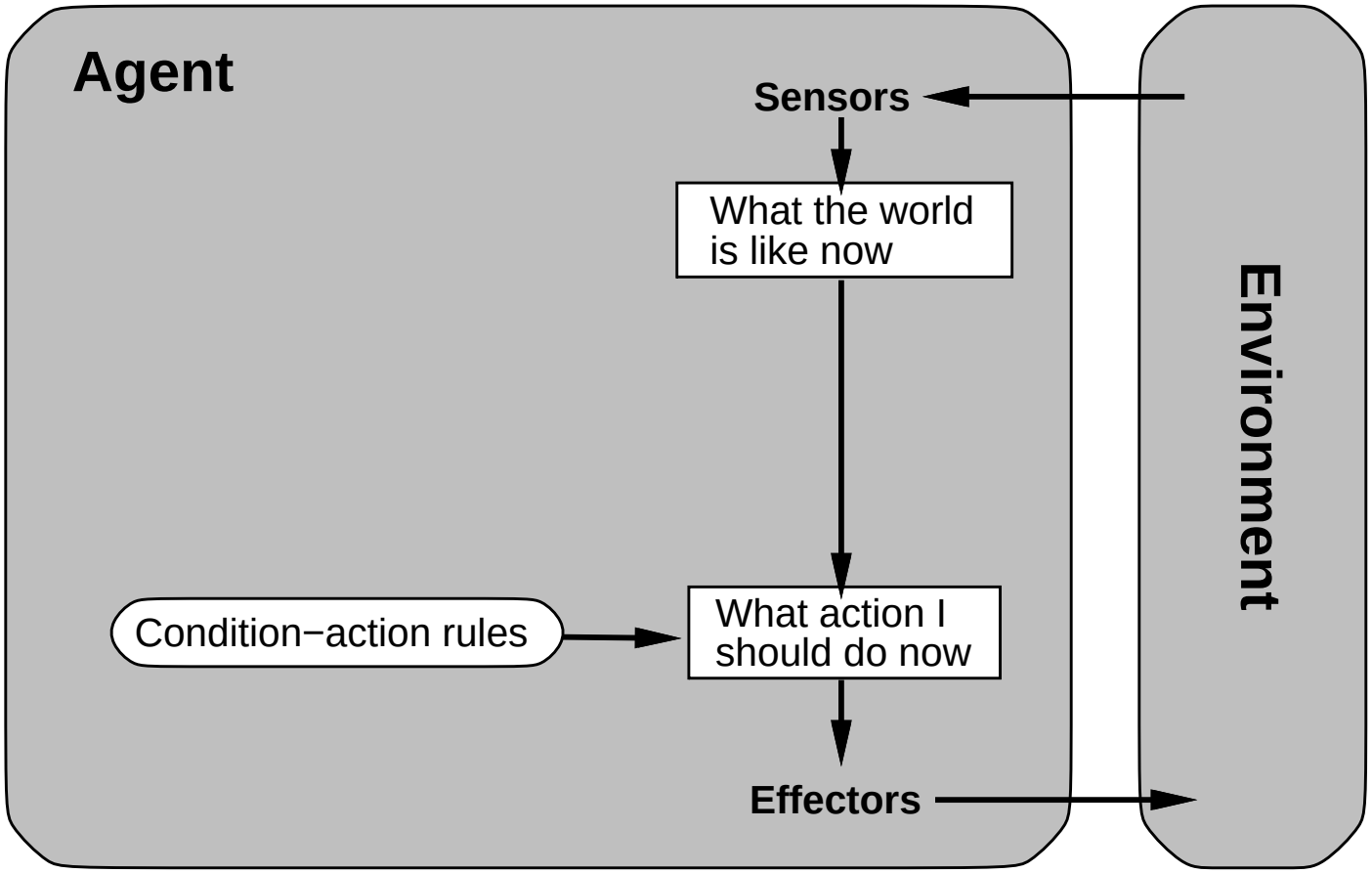
In general, agents only have incomplete/uncertain information about the environment!

We extend our framework by perceptions:

Definition 1.26 (Actions, Percepts, States)

$\mathbf{A} := \{\mathbf{a_1}, \mathbf{a_2}, \ldots, \mathbf{a_n}\}$ is the set of *actions*.
$\mathbf{P} := \{\mathbf{p_1}, \mathbf{p_2}, \ldots, \mathbf{p_m}\}$ is the set of percepts.
$\mathbf{S} := \{\mathbf{s_1}, \mathbf{s_2}, \ldots, \mathbf{s_l}\}$ is the set of *states*

Sensors don't need to provide perfect information!



Question

How can agent programs be designed?

There are four types of agent programs:

- Simple **reflex agents**
- **Agents that keep track of the world**
- Goal-based agents
- Utility-based agents

First Try

We consider a purely reactive agent and just replace states by perceptions.

Definition 1.27 (Simple Reflex Agent)

An agent is called simple reflex agent, if its function is given by

$$\mathbf{action} : \mathbf{P} \longrightarrow \mathbf{A}.$$

A Very Simple Reflex Agent

function SIMPLE-REFLEX-AGENT(*percept*) **returns** *action*
static: *rules*, a set of condition-action rules

state ← INTERPRET-INPUT(*percept*)
rule ← RULE-MATCH(*state*, *rules*)
action ← RULE-ACTION[*rule*]
return *action*

A Simple Reflex Agent with Memory

function REFLEX-AGENT-WITH-STATE(*percept*) **returns** *action*
static: *state*, a description of the current world state
rules, a set of condition-action rules

state ← UPDATE-STATE(*state, percept*)
rule ← RULE-MATCH(*state, rules*)
action ← RULE-ACTION[*rule*]
state ← UPDATE-STATE(*state, action*)
return *action*

As before, let us now consider sequences of percepts:

Definition 1.28 (Standard Agent $\mathfrak{a}$)

$$\mathbf{action} : \mathbf{P}^* \longrightarrow A$$

together with

$$\mathbf{see} : \mathbf{S} \longrightarrow \mathbf{P}.$$

An agent is thus a pair $\langle \mathbf{see}, \mathbf{action} \rangle$.

Definition 1.29 (Indistinguishable)

Two different states s, s' are indistinguishable for an agent $\mathbf{a}$, if $\mathbf{see}(s) = \mathbf{see}(s')$.

The relation "indistinguishable" on $\mathbf{S} \times \mathbf{S}$ is an **equivalence** relation.
What does $|\sim| = |\mathbf{S}|$ mean?
And what $|\sim| = 1$?

As mentioned before, the characteristic behaviour has to match with the agent design!

Definition 1.30 (Characteristic Behaviour)

The characteristic behaviour of a standard agent $\langle \mathbf{see}, \mathbf{action} \rangle$ in an environment Env is the set of all finite sequences

$$\mathbf{p_0} \rightarrow_{a_0} \mathbf{p_1} \rightarrow_{a_1} \ldots \mathbf{p_n} \rightarrow_{a_n} \ldots$$

where

$$\begin{aligned}
&\mathbf{p_0} = \mathbf{see}(\mathbf{s_0}),\\
&\mathbf{a_i} = \mathbf{action}(\langle \mathbf{p_0}, \ldots, \mathbf{p_i} \rangle),\\
&\mathbf{p_i} = \mathbf{see}(\mathbf{s_i}), \text{ where } \mathbf{s_i} \in \boldsymbol{\tau}(\mathbf{s_0}, a_0, \mathbf{s_1}, a_1, \ldots, \mathbf{s_{i-1}}, a_{i-1}).
\end{aligned}$$

Such a sequence, even if deterministic from the agent's viewpoint, may cover different environmental behaviours (runs):

$$\mathbf{s_0} \rightarrow_{a_0} \mathbf{s_1} \rightarrow_{a_1} \ldots \mathbf{s_n} \rightarrow_{a_n} \ldots$$

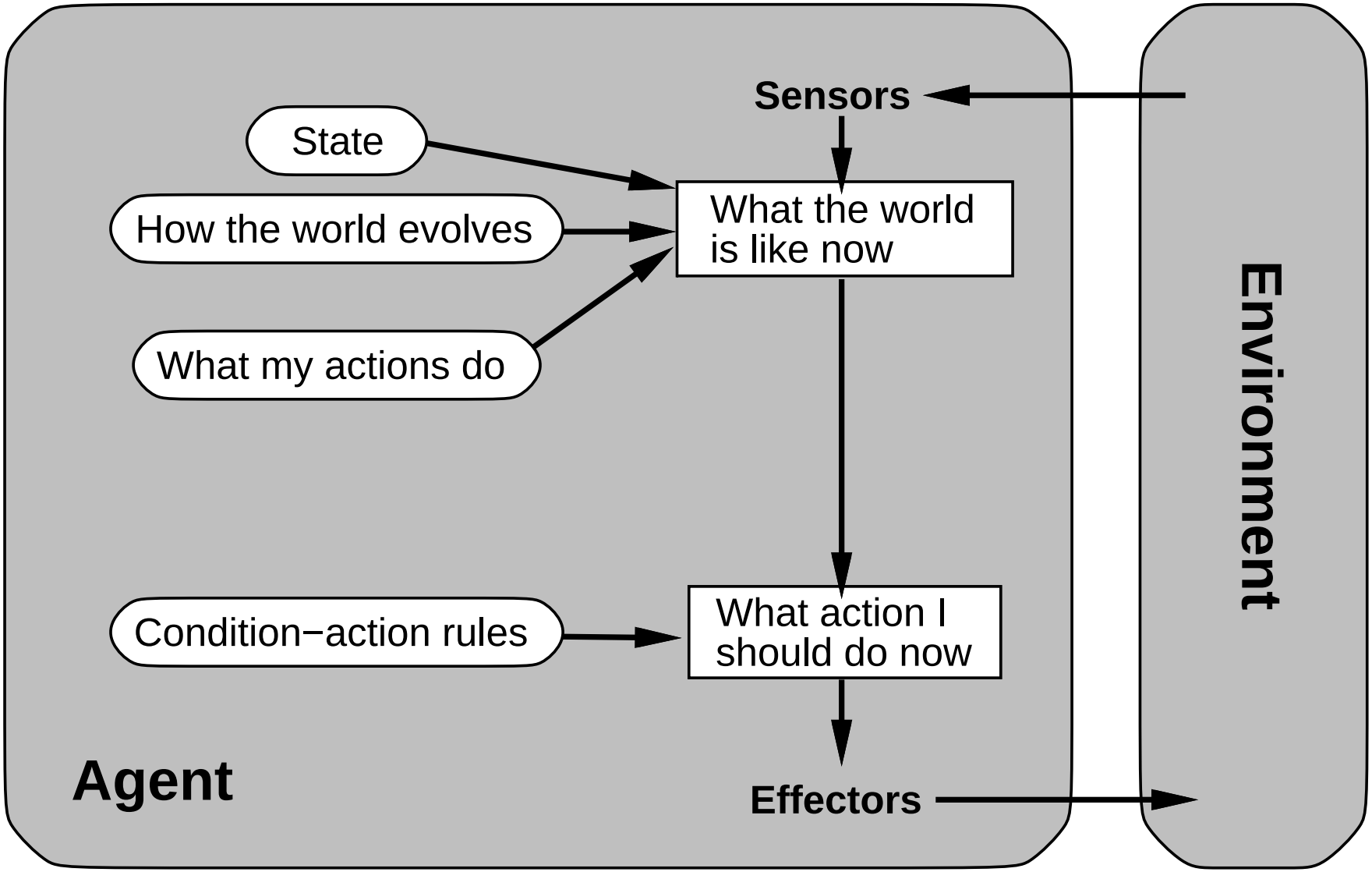
Instead of using the whole history, resp. $\mathbf{P}^*$, one can also use **internal states** $\mathbf{I} := \{\mathbf{i_1}, \mathbf{i_2}, \ldots, \mathbf{i_n}, \mathbf{i_{n+1}}, \ldots\}$.

Definition 1.31 (State-based Agent a_{state})

A state-based agent a_{state} is given by a function $\mathbf{action} : I \longrightarrow A$ together with

$$\mathbf{see} : \mathbf{S} \longrightarrow \mathbf{P},$$
$$\text{and} \quad \mathbf{next} : \mathbf{I} \times \mathbf{P} \longrightarrow \mathbf{I}.$$

Here $\mathbf{next}(\mathbf{i}, \mathbf{p})$ is the successor state of $\mathbf{i}$ if $\mathbf{p}$ is observed.


Sensors
State
How the world evolves
What the world is like now
What my actions do
Environment
Condition−action rules
What action I should do now
Agent
Effectors

Definition 1.32 (Characteristic Behaviour)

The characteristic behaviour of a state-based agent $\mathbf{a}_{state}$ in an environment Env is the set of all finite sequences

$$(\mathbf{i}_0, \mathbf{p_0}) \rightarrow_{a_0} (\mathbf{i}_1, \mathbf{p_1}) \rightarrow_{a_1} \ldots \rightarrow_{a_{n-1}} (\mathbf{i}_n, \mathbf{p_n}), \ldots$$

with

$$\mathbf{p_0} = \mathbf{see}(\mathbf{s_0}),$$
$$\mathbf{p_i} = \mathbf{see}(\mathbf{s_i}), \text{ where } \mathbf{s_i} \in \boldsymbol{\tau}(\mathbf{s_0}, a_0, \mathbf{s_1}, a_1, \ldots, \mathbf{s_{i-1}}, a_{i-1}),$$
$$\mathbf{a_n} = \mathbf{action}(\mathbf{i_{n+1}}),$$
$$\mathbf{next}(\mathbf{i_n}, \mathbf{p_n}) = \mathbf{i_{n+1}}.$$

Sequence covers the runs $\mathbf{r} : \mathbf{s_0} \rightarrow_{\mathbf{a_0}} \mathbf{s_1} \rightarrow_{\mathbf{a_1}} \ldots$ where

$$\mathbf{a_j} = \mathbf{action}(\mathbf{i_{j+1}}),$$
$$\mathbf{s_j} \in \boldsymbol{\tau}(\mathbf{s_0}, a_0, \mathbf{s_1}, a_1, \ldots, \mathbf{s_{j-1}}, a_{j-1}),$$
$$\mathbf{p_j} = \mathbf{see}(\mathbf{s_j})$$

Are state-based agents more expressive than standard agents? How to measure?

Definition 1.33 (Env. Behaviour of a_{state})

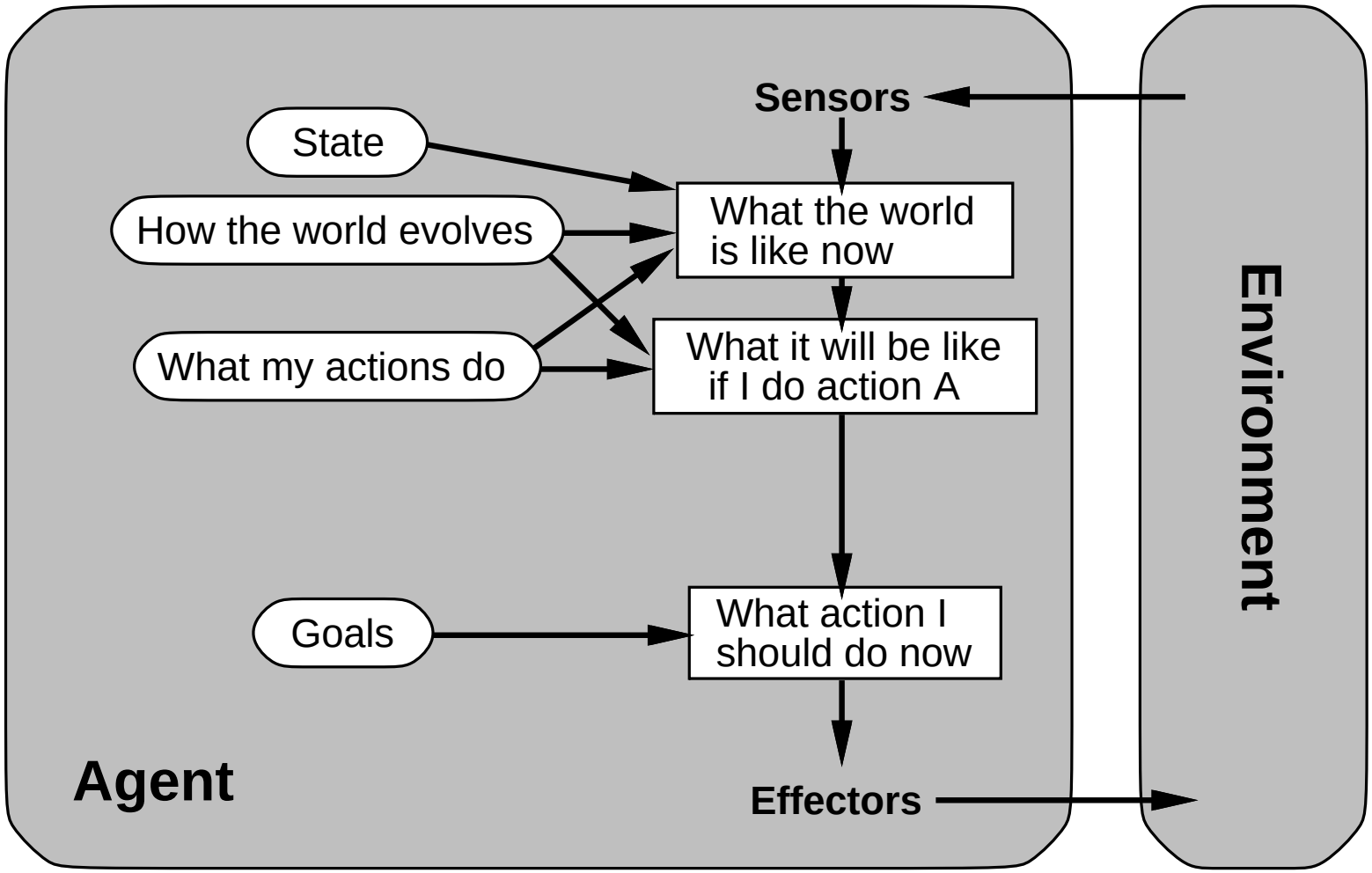
The environmental behaviour of an agent a_{state} is the set of possible runs covered by the characteristic behaviour of the agent.

Theorem 1.34 (Equivalence)

Standard agents and state-based agents are equivalent with respect to their environmental behaviour.

More precisely: For each state-based agent $\mathbf{a}_{state}$ *and* $\mathbf{next}$ *storage function there exists a standard agent* $\mathbf{a}$ *which has the same environmental behaviour, and vice versa.*

Goal based agents



This leads to **Planning**.

2. Jason

2 Jason

- Motivation
- Jason – Origins and Fundamentals
- Reasoning
- Comparison
- Advanced Features

Content of this Chapter:

We introduce a particular agent language: Jason. We

1. motivate the BDI methodology,
2. elaborate on Jason's origins and fundamentals, i.e. Jason's agent syntax,
3. explain Jason's agents semantics, i.e. the deliberation cycle, and finally
4. explain advanced features, i.e. plan failure, internal actions, possible customizations, and environment programming.

2.1 Motivation

Beliefs, Desires, Intentions (BDI)

- Model of **human practical reasoning** developed by Michael Bratman.
- **Mental attitudes**:
  - Beliefs: **Informational** state of the agent, beliefs about the world.
  - Desires: **Motivational** state of the agent, states of the world the agent wants to bring about.
  - Intentions: **Deliberative** state of the agent, what the agent has chosen to do.

Example 2.1 (Student of Computer Science)

Belief:	I am a student of computer science.
Belief:	I am in my second semester.
Desire:	Successfully attend the MAS-course.
Intention:	Visit 1st lecture
...	
Intention:	Visit 6th lecture
Intention:	Visit 1st exercise-class
...	
Intention:	Visit final exercise-class

Example 2.2 (Mental Attitudes Change)

Belief:	I am a student of computer science.
Belief:	I am in my second semester.
Desire:	Successfully attend the MAS-course.
Belief:	I can learn Jason without the lecture.
Intention:	Visit 1st exercise-class
...	
Intention:	Visit final exercise-class

2.2 Jason – Origins and Fundamentals

Agent Oriented Programming

- **Reacting** to events × **long-term** goals
- Commit to courses of action **as late as possible** and dependent of the circumstances
- **Plan failure** (dynamic environments)
- **Rational** agents
- **Social** ability
- Examples for the best known and publicly available languages
  - Jadex (Pokahr, Braubach) `http://jadex.sf.net`
  - 2APL (Dastani, Meyer) `http://www.cs.uu.nl/2apl`
  - GOAL (Hindriks)
    `http://mmi.tudelft.nl/~koen/goal.php`
  - Jason (Bordini, Hübner) `http://jason.sf.net`

AgentSpeak

- Originally proposed by Rao (1996)
- Programming language for BDI agents
- Abstract programming language aimed at theoretical results
- Elegant notation, based on logic programming
- Inspired by PRS (Georgeff & Lansky), dMARS (Kinny), and BDI Logics (Rao & Georgeff)

Jason

- Jason implements the **operational semantics** of a variant of AgentSpeak.
- Has various extensions aimed at a more **practical** programming language (e.g. definition of the MAS, communication, . . .).
- Highly customised to simplify **extensions** and **experimentation**.

Programming in Jason

Agent programming: defining the initial state(s) of agent(s)

Multi-agent programming: specifying which agents are situated in which environment (like a project-file)

Internal-actions programming: library functions that do not affect the environment (Java)

Environment programming: implementing the environment (states), external actions (state change) and percepts (from states to percepts) (Java)

Implementing agent-internals: how the agent(s) work internally, how interaction with other agents and the environment works, how the belief-base works (Java)

MAS Configuration Language I

- Simple way of defining a multi-agent system

Example 2.3 (MAS using JADE as infrastructure)

```
MAS my_system {
    infrastructure: Jade
    environment: robotEnv
    agents:
         c3po;
         r2d2 at jason.sourceforge.net;
         bob #10; // 10 instances of bob
    classpath: "../lib/graph.jar";
}
```

MAS Configuration Language II

- Configuration of event handling, frequency of perception, user-defined settings, customisations, etc.

Example 2.4 (MAS with customised agent)

```
MAS custom {
 agents: bob [verbose=2,paramters="sys.properties"]
             agentClass MyAg
             agentArchClass MyAgArch
             beliefBaseClass jason.bb.JDBCPersistentBB(
                 "org.hsqldb.jdbcDriver",
                 "jdbc:hsqldb:bookstore",
                 ...
}
```

Agent Programming (Main Concepts)

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

Goals: Represent states of affairs the agent wants to bring about

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

Intentions: Plans instantiated to achieve some goal

Agent Program

Initial belief-base: What the agent knows about the world (environment, other agents, and itself) at the beginning. **Facts** and **rules**.

Initial goal-base: Goals to achieve right from the beginning.

Plan-base: Plans are instantiated when events occur. **Sequences of actions**.

Logic Programming Excursus I

There are three concepts in logic programming:

Facts: Knowledge.

```
father(abraham,isaac).
```

Rules: To derive new knowledge.

```
man(X) :- father(X,Y).
```

Queries: Is something known (derivable)?

```
?- man(abraham).
?- man(isaac).
?- man(X).
```

Logic Programming Excursus II

Terms:

- **Constants** starting with a digit or a lower-case letter
  `abraham`, `lot`, `milcah`, `1`, `2`, `3`
- **Variables** starting with an upper-case letter
  `X`, `Y`, `List`, `Family`
- **(Compound) Terms** `f(t1, t2, ..., tn)` composed using constants, variables and functors
  `s(0)`, `s(s(0))`
- **Ground Terms** are terms without variables. They are also called **fully instantiated**.

Beliefs Representation

Syntax

Beliefs are represented by annotated literals in first order logic:

$\texttt{functor}(term_1, \ldots, term_n)\texttt{[}annot_1, \ldots, annot_m\texttt{]}$

Example 2.5 (Belief Base of Agent Tom)

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

Changes in the Belief Base I

By Perception

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

Changes in the Belief Base II

By Intention

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

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

Changes in the Belief Base III

By Communication

When an agent receives a **tell** message, the content is a new belief annotated with the sender of the message

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

Logic Programming Excursus

In logic programming there are two different negations:

Negation as failure: Anything that is neither known to be true nor derivable from the known facts using the rules in the program, is assumed to be false (`not`-operator, closed world assumption).

Strong negation: An agent explicitly believes something to be false (`~`-operator, open world assumption).

Strong Negation

Example 2.6 (Belief Base of Agent Tom)

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

Rules in Belief Base

Example 2.7

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

More later!

Goals

Types

- Achievement goal: Goal **to do**
- Test goal: Goal **to know**

Syntax

Goals has the same syntax as beliefs, but are prefixed by
! (achievement goal) or
? (test goal)

Example 2.8 (Initial Goal of Agent Tom)

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

New Goals I

By Intention

The operators ! and ? can be used to add a new goal annotated with source(self)

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

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

New Goals II

By Communication – Achieve Goal

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

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

New Goals III

By Communication – Test Goal

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

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

Events

- Events happen as a consequence to changes in the agent's beliefs or goals
- Types of events

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

- An agent reacts to events by executing plans

Plan Library

The plans that form the plan library of the agent comes from:

- initial plans defined by the programmer
- plans added dynamically and intentionally by .add_plan
  (resp. .remove_plan)
- plans received from **tellHow** messages
  (resp. **untellHow**)

Plans

Definition 2.9 (Plans)

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.

Operators for Plan's Context

Boolean operators

&	(and)
|	(or)
not	(not)
=	(unification)
>, >=	(relational)
<, <=	(relational)
==	(equals)
\ ==	(different)

Arithmetic operators

+	(sum)
-	(subtraction)
*	(multiply)
/	(divide)
div	(divide – integer)
mod	(remainder)
**	(power)

Operators for Plan's Body

A plan's body may contain:

- Goal operators (`!`, `?`, `!!`)
- Belief operators (`+`, `-`, `-+`)
- Actions and Constraints

Example 2.10 (Plan's Body)

```
+beer :  now(H) & time_to_leave(T) & H >= T
   <- !g1;           // new sub-goal
      !!g2;          // new goal
      +b1(T-H);      // add new self belief
      -+b2(T*H);     // update belief
      ?b(X);         // new sub-goal
      X > 10;        // constraint to continue the plan
      close(door).// external action
```

Example 2.11 (Plans)

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

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

Strong Negation

Example 2.12

```
+!leave(home)
   :  ~raining
   <- open(curtains); ...

+!leave(home)
   :  not raining & not ~raining
   <- .send(mum,askOne,raining,Answer,3000); ...
```

2.3 Reasoning

Syntax

Note

The Jason-documentation (`doc/Jason.pdf`) contains a precise definition of the syntax.

Semantics

Question

Now, we know the components (syntax) of agent programs. But how does the interpreter work?

Reasoning Cycle

- **How** does the Jason interpreter run an agent program?
- Jason-program = initial beliefs + initial goals + plan library
  - beliefs initialize the belief-base and generate belief-addition events,
  - goals initialize the goal-base and generate goal-addition-events.

  $\rightsquigarrow$ initial set of events
- reasoning cycle has 10 main steps

Ten Steps

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, and
10. executing one step of an intention.

Step 1: Perceiving the Environment

- Sensing the environment in order to update the belief-base,
- percepts = **list of literals** (symbolic representation of a particular property of the current state of the environment),
- `perceive`-method **default**-implementation: retrieve a list of literals of an environment implemented in Java,
- advanced: `perceive` needs to interface with real world sensors.

Step 2: Updating the Belief-Base

- **Belief update function**,
- assumption: **everything** that is currently perceivable will be included in the list of percepts,
- belief update **default**-implementation:
  1. Each percept that is not currently in the belief-base is added to the belief-base,
  2. each belief that is no longer in the percept list is removed from the belief-base,
- each change generates an event.

Step 3: Receiving Communication

- The interpreter checks the agent's mailbox,
- `checkmail` **default** implementation: process the messages in the order they were received,
- only the first message is moved to the belief-base.

Step 4: Socially Acceptable Messages

- **Socially acceptance function**: will normally need customization, possibly for each individual agent,
- **default**-implementation: accept all messages from all agents,
- advanced: cognitive reasoning about how to handle messages should be implemented in the Jason-program.

Step 5: Selecting an Event

- Practical BDI agents continually handle events,
- items are either perceived changes in the environment or changes in the agent's own goals,
- in each cycle only one pending event will be dealt with,
- **event selection function** per default selects the first event from the event-**queue** (FIFO),
- priorities can be implemented via a customized method.

Step 6: Retrieving all Relevant Plans

- Determine all plans that are relevant to the selected event,
- **relevant** plan = its triggering-event can be **unified** with the selected event.

Unification Example

Event:

```
+colour(box1,blue)[source(percept)].
```

Plans:

```
@p1 +position(Object,Coords) :  ...
@p2 +colour(Object,Colour) :  ...
@p3 +colour(Object,Colour) :  ...
@p4 +colour(Object,red) :  ...
@p5 +colour(Object,Colour)[source(self)] :  ...
@p6 +colour(Object,blue)[source(percept)] :  ...
```

Step 7: Determining Applicable Plans

- Applicable plans are plans that have a good chance in succeeding given the agent's beliefs and know-how,
- **applicable** plan = plan that is relevant + plan's context is a **logical consequence** of the belief-base.

Example 2.13 (Beliefs)

```
shape(box1,box).
pos(box1,coord(9,9)).
colour(box1,blue).
```

```
shape(sphere2,sphere).
pos(sphere2,coord(7,7)).
colour(sphere2,red).
```

Example 2.14 (Plans)

```
@p1 +colour(Object,Colour) : shape(Object,box) & not
pos(Object,coord(0,0)) <- ... .

@p2 +colour(Object,Colour): shape(Object,box) &
colour(Object,red) <- ... .
```

Step 8: Selecting One Applicable Plan

- Assumption: **Any** applicable plan will hopefully suffice for dealing with the particular selected event,
- **applicable plan selection function**: the pre-defined function selects the plan determined by the order the plans appear in the plan-library,
- an **instance** of the plan (with variable-substitutions) becomes an intention, the plan-library is not changed.

Step 9: Selecting an Intention

- An agent usually has a *set of intentions*, each representing a different focus of attention – all intentions are competing for the agent's attention,
- **intention selection function** default implementation: round-robin scheduler divides attention equally among all intentions.

Step 10: Executing One Step

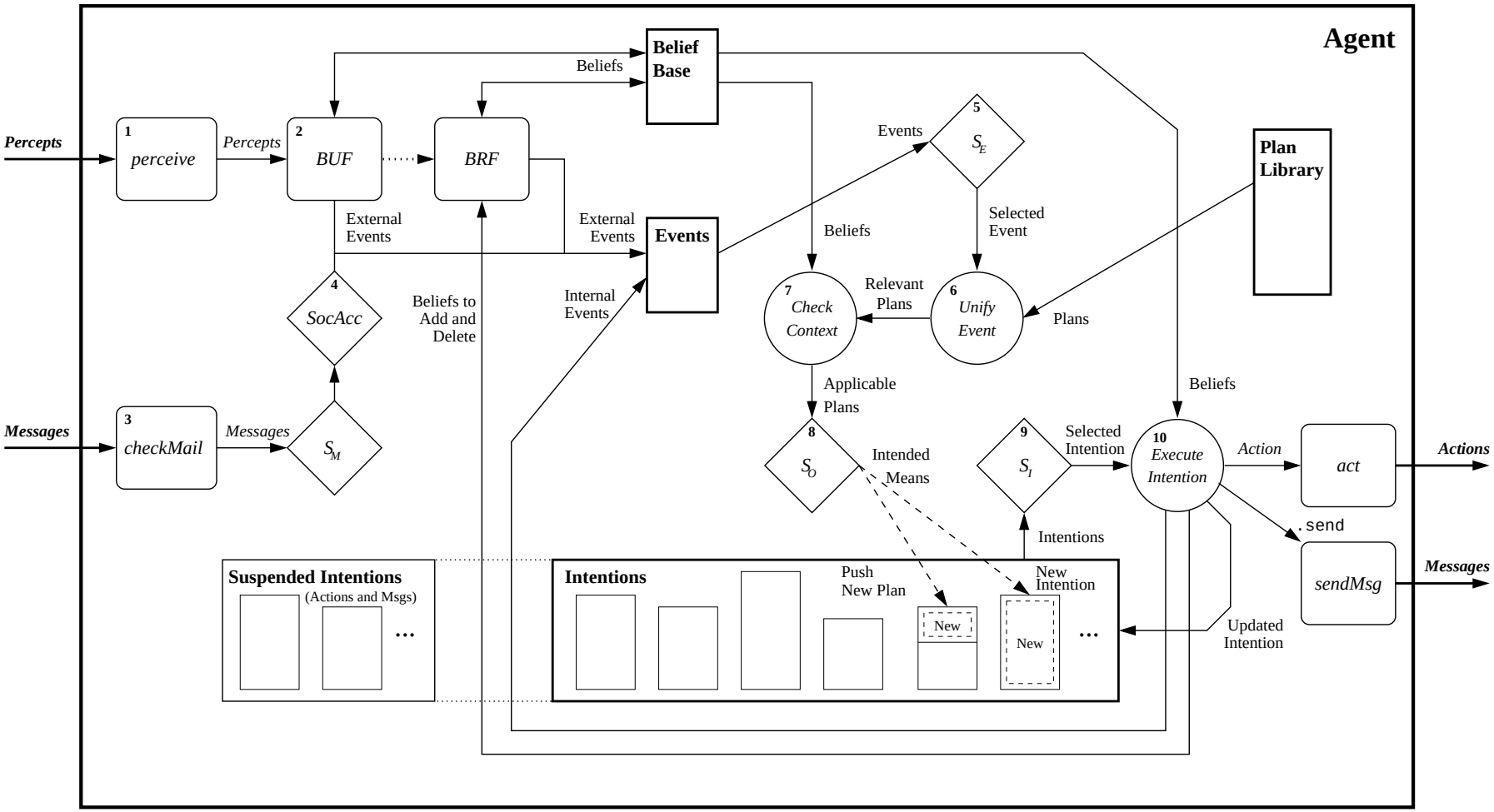
A plan is a sequence of formulæ

`f1 ; f2 ; ... ; fn .`

There are different kinds of formulæ:

- environment actions: Affect the environment, e.g. `close(door)`,
- achievement goals: Add a new goal, potentially suspend the intention, e.g. `!g1`, or `!!g2`,
- test goals: Check if a certain property is currently believed, e.g. `?b(X)`,
- mental notes: Update the belief base, e.g. `+b1(X)`, `-b2(X)`, `-+b3(X)`,
- internal actions: Execute Java-code, e.g. `pathlib.shortestPath(...)`,
- expressions: Evaluated in the usual way, e.g. `X > 10`.

Jason reasoning cycle



2.4 Comparison

Jason × Java I

Consider a very simple robot with two goals:

- When a piece of gold is seen, go for it,
- when battery is low, charge it.

Example 2.15 (Java Code – Go to Gold)

```
public class Robot extends Thread {
   boolean seeGold, lowBattery;
   public void run() {
      while (true) {
          while (!  seeGold) {}
          while (seeGold) {
              a = selectDirection();
              doAction(go(a));
} } } }
```

(How to code the charge battery behaviour?)

Jason × Java II

Example 2.16 (Java Code – Charge Battery)

```
public class Robot extends Thread {
   boolean seeGold, lowBattery;
   public void run() {
      while (true) {
          while (!  seeGold)
              if (lowBattery) charge();
          while (seeGold) {
              a = selectDirection ();
              if (lowBattery) charge();
              doAction(go(a));
              if (lowBattery) charge();
} } } }
```

(Note where the test for low battery have to be done!)

Jason × Java III

Example 2.17 (Jason Code)

```
+see(gold)
   <- !goto(gold).
+!goto(gold) :  see(gold)
   <- !select_direction(A);
      go(A);
      !goto(gold).
+battery(low)
   <- .suspend(goto(gold));
      !charge;
      .resume(goto(gold)).
```

Jason × Prolog

- With the Jason extensions, nice separation of theoretical and practical reasoning.
- BDI architecture 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.

2.5 Advanced Features

Plan Failure

Several reasons for plan failure:

- Lack of relevant or applicable plans for an achievement goal: Agent does not know **how** to achieve something desired. A subgoal cannot be achieved.
- Failure of a test goal: Represents a situation where the agent is expected to believe that a certain property is true, but it was not. First try to get the information from the belief-base, than try to acquire it by instantiating a plan. If both fail then the plan fails.
- Action failure: Internal actions and external actions can fail.

What Happens When a Plan Fails?

- Regardless of the reason for plan failing, the interpreter generates a **goal deletion event**.
- Plans for goal deletion events = **clean-up** plans.

Example 2.18 (Plan Failure)

```
!g1. // initial goal
@p1 +!g1 : true <- !g2(X); .print(”end g1 ”,X) .
@p2 +!g2(X) : true <- !g3(X); .print(”end g2 ”,X) .
@p3 +!g3(X) : true <- !g4(X); .print(”end g3 ”,X) .
@p4 +!g4(X) : true <- !g5(X); .print(”end g4 ”,X) .
@p5 +!g5(X) : true <- .fail .
@f1 -!g3(failure) : true <- .print(”in g3 failure”)
.
```

Output:

```
[a] saying: in g3 failure
[a] saying: end g2 failure
[a] saying: end g1 failure
```

Internal Actions

- Unlike actions, internal actions do not change the environment.
- Code to be executed as part of the agent reasoning cycle.
- AgentSpeak is meant as a high-level language for the agent's practical reasoning and internal actions can be used for invoking legacy code elegantly.

- Internal actions can be defined by the user in Java

```
libname.action_name(...)
```

Standard Internal Actions

- Standard (pre-defined) internal actions have an empty library name
  - `.print`$(term_1, term_2, \ldots)$
  - `.union`$(list_1,\ list_2,\ list_3)$
  - `.my_name`(var)
  - `.send`$(ag, perf, literal)$
  - `.intend`$(literal)$
  - `.drop_intention`$(literal)$
- Many others available for: printing, sorting, list/string operations, manipulating the beliefs/annotations/plan library, creating agents, waiting/generating events, etc. $\rightsquigarrow$ documentation.

Possible Customisations in Jason

- **Agent** class customisation:
  selectMessage, selectEvent, selectOption,
  selectIntetion, buf, brf, ...
- Agent **architecture** customisation:
  Perceive, act, sendMsg, checkMail, ...
- **Belief base** customisation:
  Add, remove, contains, ...
  - Example: Persistent belief base
    (in text files, in data bases, ...)

Communication Infrastructure

Different communication and execution management infrastructures can be used with Jason:

Centralised: All agents in the same machine, one thread by agent, very fast.

Centralised (pool): All agents in the same machine, fixed number of thread, allows thousands of agents.

Jade: Distributed agents, FIPA-ACL.

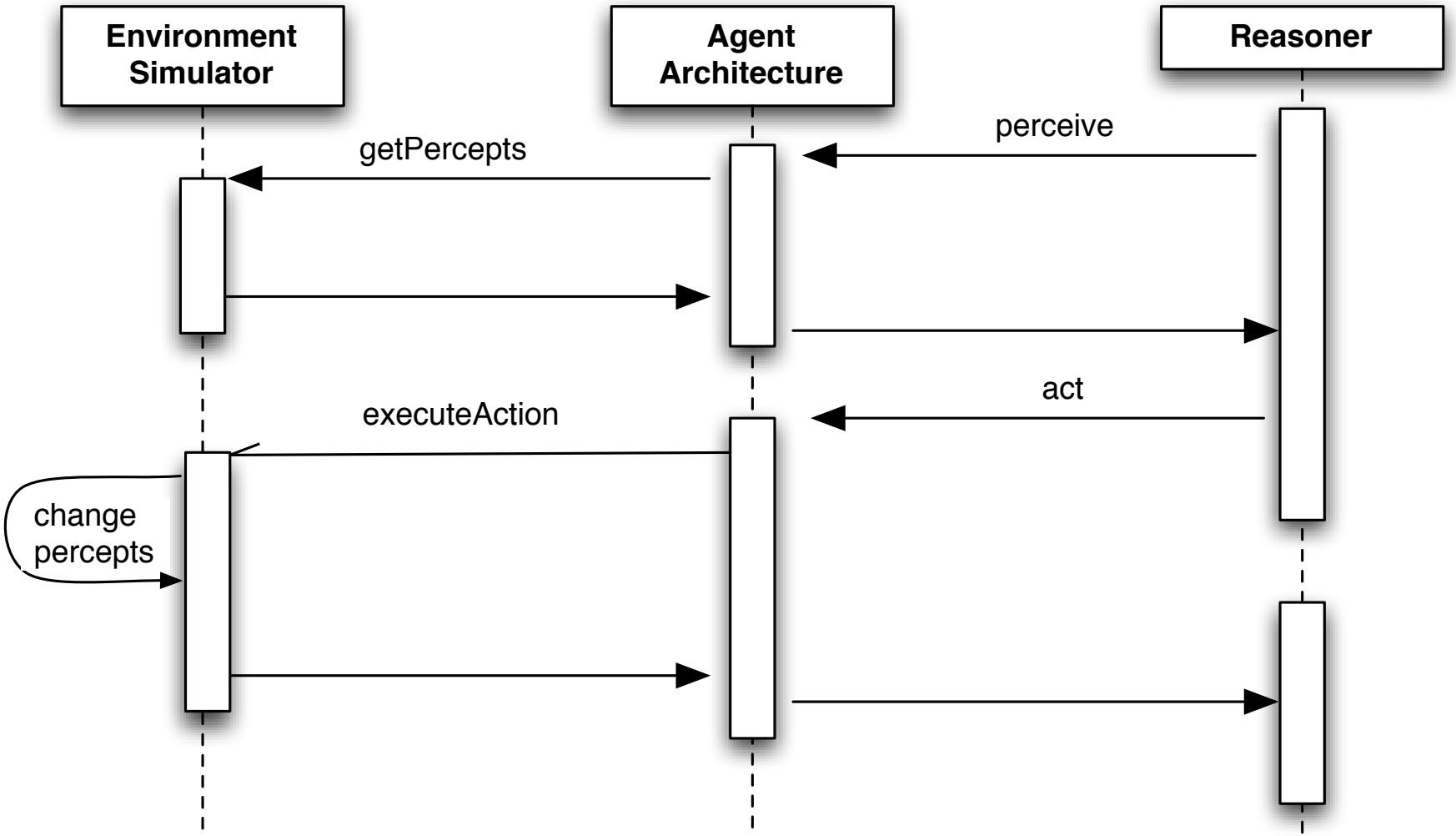
Saci: Distributed agents, KQML.

.... Others defined by the user (e.g. AgentScape)

Definition of a Simulated Environment

- Normally, there will be an environment where the agents are situated.
- The agent architecture needs to be customised to get perceptions and act on such environment.
- We often want a simulated environment (e.g. to test a MAS application).
- This is done in Java by extending Jason's Environment class.

Interplay with Environment Simulator



Example of an Environment Class

```
1 import jason.*;
2 import ...;
3 public class robotEnv extends Environment {
4   ....
5   public robotEnv() {
6     Literal gp =
7           Literal.parseLiteral("green_patch(souffle)");
8     addPercept(gp);
9   }
10
11  public boolean executeAction(String ag, Structure action) {
12    if (action.equals(...)) {
13      addPercept(ag,
14           Literal.parseLiteral("location(souffle,c(3,4))");
15    }
16    ...
17    return true;
18 } }
```

3. Jason Logic Programming

Content of this Chapter:

We explain how you can program the belief-base of a Jason-agent by means of logic programming. We

1. show how you can define and make use of facts and rules,
2. how you can employ recursion in order to implement complex functions, and
3. elaborate on lists, which is a useful data-structure.

Remember

- Jason is based on logic programming.
- Belief-base: **facts + rules**.
- **Queries** in Jason: Contexts of plans and test-goals.

```
+!start : father(abraham,issac) <-
    ?male(abraham).
```

Now

Short Introduction to Logic Programming

- Biblical Research,
- arithmetics
- lists.

3.1 Biblical Research

Some Facts

```
father(terach,abraham).
father(terach,nachor).
father(terach,haran).
father(abraham,isaac).
father(haran,lot).
father(haran,milcah).
father(haran,yiscah).

mother(sarah,isaac).
```

```
male(terach).
male(abraham).
male(nachor).
male(haran).
male(isaac).
male(lot).

female(sarah).
female(milcah).
female(yiscah).
```

How to Define “son” and “daughter”?

`son(X,Y)` means `X` is a son of `Y`. `daughter(X,Y)` means `X` is a daughter of `Y`.

```
son(X,Y) :- father(Y,X) & male(X).
son(X,Y) :- mother(Y,X) & male(X).
daughter(X,Y) :- father(Y,X) & female(X).
daughter(X,Y) :- mother(Y,X) & female(X).
```

How to Define "grandparent"?

`grandparent(X,Y)` means `X` is a grandparent of `Y`.

Solution 1:

```
grandparent(X,Y) :- father(X,Z) & father(Z,Y).
grandparent(X,Y) :- father(X,Z) & mother(Z,Y).
grandparent(X,Y) :- mother(X,Z) & father(Z,Y).
grandparent(X,Y) :- mother(X,Z) & mother(Z,Y).
```

Solution 2:

```
parent(X,Y) :- father(X,Y).
parent(X,Y) :- mother(X,Y).
grandparent(X,Y) :- parent(X,Z) & parent(Z,Y).
```

Giving More Meaning to the Variables

```
parent(Parent,Child) :- father(Parent,Child).
parent(Parent,Child) :- mother(Parent,Child).
```

More Examples

```
procreated(Man,Woman) :- father(Man,Child) &
                         mother(Woman,Child).

sibling(Sib1,Sib2) :- parent(Parent,Sib1) &
                      parent(Parent,Sib2).
```

$\rightsquigarrow$ **Problem!**

```
sibling(Sib1,Sib2) :- parent(Parent,Sib1) &
                      parent(Parent,Sib2) &
                      Sib1 \== Sib2.
```

Even More Examples

```
cousin(Cousin1,Cousin2) :- parent(Parent1,Cousin1) &
                           parent(Parent2,Cousin2) &
                           sibling(Parent1,Parent2).

uncle(Uncle,Person) :- brother(Uncle,Parent) &
                       parent(Parent,Person).
```

How to Define “ancestor”?

Approach 1:

```
grandparent(Ancestor,Descendant) :-
  parent(Ancestor,Person) &
  parent(Person,Descendant).
greatgrandparent(Ancestor,Descendant) :-
  parent(Ancestor,Person) &
  grandparent(Person,Descendant).
greatgreatgrandparent(Ancestor,Descendant) :-
  parent(Ancestor,Person) &
  greatgrandparent(Person,Descendant).
```

How to Define “ancestor”?

Approach 2 (more elegant using recursion):

```
ancestor(Ancestor,Descendant) :-
  parent(Ancestor,Descendant).

ancestor(Ancestor,Descendant) :-
  parent(Ancestor,Person) &
  ancestor(Person,Descendant).
```

3.2 Arithmetics

Representing $\mathbb{N}$ with Terms

Using a single constant 0 and the successor-function $\mathtt{s}$

Natural number	Term representation
0	0
1	$\mathtt{s(0)}$
2	$\mathtt{s(s(0))}$
3	$\mathtt{s(s(s(0)))}$
4	$\mathtt{s(s(s(s(0))))}$
$\ldots$	$\ldots$
n	$\mathtt{s}^n(0)$

Checking

Question

How can we determine whether an arbitrary term is a natural number or not?

```
natural_number(0).
natural_number(s(X)) :- natural_number(X).
```

How to Add Natural Numbers?

`plus(X,Y,Z)`, where `Z` is the sum of `X` and `Y`.

```
plus(0,X,X) :- natural_number(X).
plus(s(X),Y,s(Z)) :- plus(X,Y,Z).
```

The definition is not unique:

```
plus(0,X,X) :- natural_number(X).
plus(X,s(Y),s(Z)) :- plus(X,Y,Z).
```

works fine, too.

How Can this be Used?

Is the sum correct?

`?plus(s(0),s(0),s(s(0)))` $\rightsquigarrow$ **true**

What is the sum of two numbers?

`?plus(s(0),s(0),Z)` $\rightsquigarrow$ `Z` **is** `s(s(0))`

What is the difference of two numbers?

`?plus(s(0),Y,s(s(0)))` $\rightsquigarrow$ `Y` **is** `s(0)`

How to Multiply Natural Numbers?

`times(X,Y,Z)`, where `Z` is the product of `X` and `Y`.

Multiplication as repeated addition:

```
times(0,X,0) :- natural_number(X).
times(s(X),Y,Z) :- times(X,Y,XY) &
                   plus(XY,Y,Z).
```

Do You Do Maths Like this in Jason?

Fortunately **not**: arithmetics is implemented on a level underneath the logic programming level (system predicates).

Is the sum correct?

`1 + 1 = 2` $\rightsquigarrow$ **true**

What is the sum of two numbers?

`1 + 1 = Z` $\rightsquigarrow$ `Z` **is** `2`

What is the difference of two numbers?

`Y = 2 - 1` $\rightsquigarrow$ `Y` **is** `1`

3.3 Lists

What are Lists?

- special and useful **terms**
- can be used to define complex **data-structures** (arrays, tables, tensors, trees, ...)

Examples:

`[]`	the empty-list
`[a]`	a list with one element
`[1,2,3,4]`	a couple of entries
`[[1,2,3],[4,5,6],[7,8,9]]`	a matrix
`[node(0,0),node(0,1),node(1,1)]`	a path

List Operator

There is a list operator that separates a list's head from its tail. Example:

```
[a,b,c] = [H|T]
```

will yield this substitution

```
H -> a
T -> [b,c]
```

Examples

Printing the head of a list:

```
!printHead([a,b,c]).
+!printHead([H|T]) : true <- .print(H).
```

yields

```
[agent] a
```

Examples

Printing the tail of a list:

```
!printTail([a,b,c]).
+!printTail([H|T]) : true <- .print(T).
```

yields

```
[agent] [b,c]
```

Examples

Printing a list recursively:

```
!printHeadRec([a,b,c]).
+!printHeadRec([H|T]) : true <-
    .print(H);
    !printHeadRec(T).
+!printHeadRec([]).
```

yields

```
[agent] a
[agent] b
[agent] c
```

Testing for Membership

`member(X,List)` is true iff `X` is a member of `List`

```
member(X,[X,List]).
member(X,[HT]) :- member(X,T).
```

Appending Two Lists

`append(X,Y,Z)` is true iff `Z` is the concatenation of `X` and `Y`.
E.g. `append([a,b],[c,d],[a,b,c,d]).`

```
append([],Ys,Ys).
append([X|Xs],Ys,[X|Zs]) :-
  append(Xs,Ys,Zs).
```

Built-in Internal Actions

`.concat(X,Y,Z)` appends two lists

`.member(X,List)` checks if an element is a member of a list

`.length(List,L)` yields the length of a list

`.delete(X,List,NewList)` deletes an lement from a list

`.reverse(List,NewList)` reverses a list

`.nth(N,List,X)` yields the n-th element of a list

`.union(S1,S2,NewSet)` yields the union of two sets

`.intersection(S1,S2,NewSet)` yields the intersection of two sets

...

$\rightsquigarrow$ more in the documentation