Multiagent Systems I

Prof. Dr. Jürgen Dix

Department of Informatics
Clausthal University of Technology
WS 09/10
**Time:** Monday, Tuesday: 10–12

**Place:** Am Regenbogen, IfI (Lab)

**Labs:** From 3. November on.

**Website**

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

Visit regularly!

There you will find important information about the lecture, documents, labs et cetera.

**Lecture:** Prof. Dix

**Labs:** T. Behrens, M. Köster

**Exam:** Demos at the end of the semester.
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 fight against each other. The winning team will be determined in a competition and get a price.

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.

Lecture Overview

1. Week: 1. Introduction, 2. Basics
2. Week: 3. Scenarios, 4. Jason
Outline

1. Introduction
2. Basic Notions
3. Some Scenarios
4. Jason
1. Introduction

- Why Agents?
- Intelligent Agents
- Formal Description
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?
1 Introduction

1.1 Why Agents?

Definition 1.1 (Distributed Artificial Intelligence (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.

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 (1))

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

*Figure: Grand Challenge 2004*
Example 1.10 (Grand Challenge 2004 (2))

- 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 1.11 (Grand Challenge 2005 (1))

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

*Figure:* VW Touareg coached by Stanford University
Example 1.12 (Grand Challenge 2005 (2))

- 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 1.13 (Urban Challenge (1))

**Urban Challenge**: Organised by DARPA since 2007.

**Figure**: Urban Challenge 2007
Example 1.14 (Urban Challenge (2))

- 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 1.15 (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
Example 1.16 (CLIMA Contest: Gold Mining)

Figure: Gold Mining 2006: CLIMABot (blue) vs. brazil (red)
Example 1.17 (Agent Contest: Chasing Cows)

Second try: *Push cows in a corral.*

- [http://multiagentcontest.org](http://multiagentcontest.org)
Example 1.18 (Agent Contest: Chasing Cows)

Figure: Chasing Cows 2008
Example 1.19 (Agent Contest: Chasing Cows)

Figure: Chasing Cows 2009
Agents: Why do we need them?

Information systems are distributed, open, heterogeneous. 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
1 Introduction

1.1 Why Agents?

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.
<table>
<thead>
<tr>
<th>AI</th>
<th>DAI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agent</td>
<td><strong>Multiple</strong> Agents</td>
</tr>
<tr>
<td>Intelligence:</td>
<td>Intelligence:</td>
</tr>
<tr>
<td>Property of a <strong>single</strong> Agent</td>
<td>Property of <strong>several</strong> Agents</td>
</tr>
<tr>
<td><strong>Cognitive</strong> Processes of a <strong>single</strong> Agent</td>
<td><strong>Social</strong> Processes of <strong>several</strong> Agents</td>
</tr>
</tbody>
</table>
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 1.20 (Agent $a$)

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

![Diagram showing the relationship between an agent, environment, sensors, percepts, actions, and effectors.](image)
Definition 1.21 (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 1.22 (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:

set of percept sequences $\rightarrow$ 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
**Table: Examples of agents types and their PEAS descriptions.**

<table>
<thead>
<tr>
<th>Agent Type</th>
<th>Perform. Measure</th>
<th>Environment</th>
<th>Actuators</th>
<th>Sensors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Medical diagnosis system</td>
<td>Healthy patient, minimize costs</td>
<td>Patient, hospital, staff</td>
<td>Display questions, tests, diagnoses, treatments</td>
<td>Entry of symptoms, findings, patient’s answers</td>
</tr>
<tr>
<td>Satellite image analysis system</td>
<td>Correct image categorization</td>
<td>Downlink from orbiting satellite</td>
<td>Display categorization of scene</td>
<td>Color pixel arrays</td>
</tr>
<tr>
<td>Part-picking robot</td>
<td>Percentage of parts in correct bins</td>
<td>Conveyor belt with parts; bins</td>
<td>Jointed arm and hand</td>
<td>Camera, joint angle sensors</td>
</tr>
<tr>
<td>Interactive English tutor</td>
<td>Maximize student’s score on test</td>
<td>Set of students, testing agency</td>
<td>Display exercises, suggestions, corrections</td>
<td>Keyboard entry</td>
</tr>
</tbody>
</table>
**Question:**
How do environment properties influence agent design?

**Definition 1.23 (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.
<table>
<thead>
<tr>
<th>Environment</th>
<th>Accessible</th>
<th>Deterministic</th>
<th>Episodic</th>
<th>Static</th>
<th>Discrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chess with a clock</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Semi</td>
<td>Yes</td>
</tr>
<tr>
<td>Chess without a clock</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Poker</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Backgammon</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Taxi driving</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Medical diagnosis system</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Image-analysis system</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Semi</td>
<td>No</td>
</tr>
<tr>
<td>Part-picking robot</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Refinery controller</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Interactive English tutor</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
xbiff, software demons are agents (not intelligent).

**Definition 1.24 (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

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 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:

```plaintext
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
```
An agent example – taxi driver:

<table>
<thead>
<tr>
<th>Agent Type</th>
<th>Perform. Measure</th>
<th>Environment</th>
<th>Actuators</th>
<th>Sensors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Taxi driver</td>
<td>Safe, fast, legal, maximize profits</td>
<td>Roads, other traffic, pedestrians, customers</td>
<td>Steering, accelerator, brake, signal, horn</td>
<td>Cameras, sonar, GPS odometer, engine sensors</td>
</tr>
</tbody>
</table>

Table: PEAS description of the task environment for an automated taxi
Some examples:

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

```javascript
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

\[ A := \{a_1, \ldots, a_n\} \quad \text{(set of actions)} \]

and change the state of the environment

\[ S := \{s_1, s_2, \ldots, s_n\} \quad \text{(set of states)} \]
How does the environment (the state $s$) develop when an action $a$ is executed?

We describe this with a function

$$\text{env} : S \times A \rightarrow 2^S.$$ 

This includes non-deterministic environments.
How do we describe agents?

We could take a function \( \text{action} : S \rightarrow A \).
Question:
How can we describe an agent, now?

Definition 1.25 (Purely Reactive Agent)
An agent is called purely reactive, if its function is given by

\[ \text{action} : S \rightarrow A. \]
This is too weak!

Take the whole history (of the environment) into account: \( s_0 \rightarrow a_0 \ s_1 \rightarrow a_1 \ \ldots \ s_n \rightarrow a_n \ \ldots \)

The same should be done for env!
This leads to agents that take the whole sequence of states into account, i.e.

\[ \text{action} : S^* \rightarrow A. \]

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 1.26 (Run \( r, R = R^{act} \cup R^{state} \))**

A run \( r \) over \( A \) and \( S \) is a finite sequence

\[
    r : s_0 \rightarrow a_0 \ s_1 \rightarrow a_1 \ \ldots s_n \rightarrow a_n \ \ldots
\]

Such a sequence may end with a state \( s_n \) or with an action \( a_n \): 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 1.27 (Environment, 2. version)

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

1. the set $S$ of states,
2. the initial state $s_0 \in S$,
3. a function $\tau : \mathcal{R}^{act} \rightarrow 2^S$, which describes how the environment changes when an action is performed (given the whole history).
Definition 1.28 (Agent α)

An agent α is determined by a function

\[
\text{action} : R^{state} \rightarrow A,
\]

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

Important:

An agent system is then a pair \( α = \langle \text{action}, Env \rangle \) consisting of an agent and an environment. We denote by \( R(α, Env) \) the set of runs of agent α in environment Env.
Definition 1.29 (Characteristic Behaviour)

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

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

For deterministic $\tau$, 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 $a, b$ are called **behaviourally equivalent wrt. environment** $Env$, if $R(a, Env) = R(b, Env)$.

Two agents $a, b$ 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 1.30 (Actions, Percepts, States)

\[ A := \{a_1, a_2, \ldots, a_n\} \] is the set of actions.
\[ P := \{p_1, p_2, \ldots, p_m\} \] is the set of percepts.
\[ S := \{s_1, s_2, \ldots, s_l\} \] is the set of states
Sensors don’t need to provide perfect information!

Agent

Environment

Sensors

What the world is like now

Condition–action rules

What action I should do now

Effectors
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 1.31 (Simple Reflex Agent)

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

\[
\text{action} : P \rightarrow A.
\]
A very simple reflex agent

```plaintext
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(\textit{percept}) returns \textit{action}

\textbf{static}: state, a description of the current world state
\textit{rules}, a set of condition-action rules

\textit{state} \leftarrow \textsc{Update-State}(state, \textit{percept})
\textit{rule} \leftarrow \textsc{Rule-Match}(state, \textit{rules})
\textit{action} \leftarrow \textsc{Rule-Action}[\textit{rule}]
\textit{state} \leftarrow \textsc{Update-State}(state, \textit{action})

\textbf{return} \textit{action}
As before, let us now consider sequences of percepts:

**Definition 1.32 (Standard Agent α)**

\[ \text{action} : P^* \rightarrow A \]

\[ \text{together with} \]

\[ \text{see} : S \rightarrow P. \]

An agent is thus a pair \( \langle \text{see}, \text{action} \rangle \).
Definition 1.33 (Indistinguishable)

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

The relation “indistinguishable” on $S \times S$ is an equivalence relation.

What does $|\sim| = |S|$ mean?
And what $|\sim| = 1$?

As mentioned before, the characteristic behaviour has to match with the agent design!
**Definition 1.34 (Characteristic Behaviour)**

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

\[
p_0 \rightarrow a_0 \quad p_1 \rightarrow a_1 \quad \ldots \quad p_n \rightarrow a_n \quad \ldots
\]

where

\[
p_0 = \text{see}(s_0), \quad a_i = \text{action}(\langle p_0, \ldots, p_i \rangle), \quad p_i = \text{see}(s_i), \quad \text{where } s_i \in \tau(s_0, a_0, s_1, a_1, \ldots, s_{i-1}, a_{i-1}).
\]

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

\[
S_0 \rightarrow a_0 \quad S_1 \rightarrow a_1 \quad \ldots \quad S_n \rightarrow a_n \quad \ldots
\]
Instead of using the whole history, resp. $P^*$, one can also use \textbf{internal states} $I := \{i_1, i_2, \ldots, i_n, i_{n+1}, \ldots\}$.

\textbf{Definition 1.35 (State-based Agent $a_{\text{state}}$)}

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

$$\text{see}: S \rightarrow P,$$

and $$\text{next}: I \times P \rightarrow I.$$ 

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

1.3 Formal Description

Agent

Environment

Sensors

Effectors

What the world is like now

What action I should do now

How the world evolves

What my actions do

State

Condition–action rules
Definition 1.36 (Characteristic Behaviour)

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

$$(i_0, p_0) \rightarrow a_0 (i_1, p_1) \rightarrow a_1 \ldots \rightarrow a_{n-1} (i_n, p_n), \ldots$$

with

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

Sequence covers the runs $r : s_0 \rightarrow a_0 s_1 \rightarrow a_1 \ldots$ where

$a_j = \text{action}(i_{j+1}),$
$s_j \in \tau(s_0, a_0, s_1, a_1, \ldots, s_{j-1}, a_{j-1}),$
$p_j = \text{see}(s_j)$
Are state-based agents more expressive than standard agents? How to measure?

**Definition 1.37 (Env. Behaviour of $a_{\text{state}}$)**

The *environmental behaviour* of an agent $a_{\text{state}}$ is the set of possible runs covered by the characteristic behaviour of the agent.
Theorem 1.38 (Equivalence)

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

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

This leads to **Planning**.
2. Basic Notions

- Reactive Agents
- BDI-Architecture
- PROLOG
Content of this chapter:

In this chapter we present some important techniques that will be used later for programming agents.

- An architecture for reactive agents, based on a subsumption.
- The BDI/Agent oriented programming-, architecture. While Jason is not exactly based on this version of BDI, it is very similar in spirit.
- We introduce some PROLOG technology: terms, facts and rules. These are the basic ingredients for writing agents in the labs.
2.1 Reactive Agents
Idea:

Intelligent behaviour is **Interaction of the agents with their environment.**

It emerges through splitting in simpler interactions.
Subsumption-Architectures

- Decision making is realized through goal-directed behaviours: each behaviour is an individual action. nonsymbolic implementation.
### Formal Model

- **see**: as up to now, but close relation between observation and action: **no transformation of the input**.

- **action**: Set of behaviors and inhibition relation.

  \[ Beh := \{ \langle c, a \rangle : c \subseteq P, a \in A \} \]

  \( \langle c, a \rangle \) “fires” if \( \text{see}(s) \in c \) (\( c \) stands for “condition”).

  \( \prec \subseteq A_{rules} \times A_{rules} \)

  is called inhibition-relation, \( A_{rules} \subseteq Beh \).

  We require \( \prec \) to be a total ordering.

  \( b_1 \prec b_2 \) means: \( b_1 \) inhibits \( b_2 \),

  \( b_1 \) has priority over \( b_2 \).
Function: Action Selection in the Subsumption Architecture

1. function action(p:P):A
2. var fired:℘(R)
3. var selected:A
4. begin
5. \[ fired \leftarrow \{(c,a) \mid (c,a) \in R \text{ and } p \in c\} \]
6. for each \((c,a) \in fired\) do
7. \[ \text{if } \neg(\exists (c',a') \in fired \text{ such that } (c',a') < (c,a)) \text{ then} \]
8. return a
9. end-if
10. end-for
11. return null
12. end function action

Figure 5.1 Action Selection in the subsumption architecture.
Example 2.1 (Exploring a Planet)

A distant planet (asteroid) is assumed to contain gold. Samples should be brought to a spaceship landed on the planet. It is not known where the gold is. Several autonomous vehicles are available. Due to the topography of the planet there is no connection between the vehicles.

The spaceship sends off radio signals: gradient field.
Low Level Behaviour:

1. If detect an obstacle then change direction.

2. Layer:

2.1 If Samples on board and at base then drop off.

2.2 If Samples on board and not at base then follow gradient field.

3. Layer:

3.1 If Samples found then pick them up.

4. Layer:

4.1 If true then take a random walk.

With the following ordering:

(1) ≺ (2) ≺ (3) ≺ (4) ≺ (5).

Under which assumptions (on the distribution of the gold) does this work perfectly?
Vehicles can *communicate indirectly* with each other:

- they put off, and
- pick up

*radiactive samples* that can be sensed.
Low Level Behaviour:

1. (1) If detect an obstacle then change direction.

2. Layer:

2. (2) If Samples on board and at base then drop off.
2. (3) If Samples on board and not at base then drop off two radioactive crumbs and follow gradient field.

3. Layer:

3. (4) If Samples found then pick them up.
3. (5) If radioactive crumbs found then take one and follow the gradient field (away from the spaceship).

4. Layer:

4. (6) If true then take a random walk.

With the ordering (1) < (2) < (3) < (4) < (5) < (6).
**Pro:** Simple, economic, efficient, robust, elegant.

**Contra:**

- Without knowledge about the environment agents need to know about the own local environment.
- Decisions only based on local information.
- How about bringing in **learning**?
- Relation between agents, environment, and behaviours is not clear.
- Agents with \( \leq 10 \) behaviours are doable. But the more layers the more complicated to understand what is going on.
2.2 BDI-Architecture
Belief, Desire, Intention.

Agent Control Loop Version 1
1. while true
2. observe the world;
3. update internal world model;
4. deliberate about what intention to achieve next;
5. use means-ends reasoning to get a plan for the intention;
6. execute the plan
7. end while
Agent Control Loop Version 2
1. \( B := B_0; \) /* initial beliefs */
2. while true do
3. get next percept \( \rho; \)
4. \( B := brf(B, \rho); \)
5. \( I := deliberate(B); \)
6. \( \pi := plan(B, I); \)
7. \( \text{execute}(\pi) \)
8. end while
Three main questions:

**Deliberation:** How to deliberate?

**Planning:** Once committed to something, how to reach the goal?

**Replanning:** What if during execution of the plan, things are running out of control and the original plan fails?
Belief 1: Making money is important.
Belief 2: I like computing.
Desire 1: Graduate in Computer Science.
Desire 2 (Int.): Pass the BSc.
Desire 3: Graduate in time, marks are unimportant.
New Belief: Money is not so important after all.
New Belief: Working scientifically is fun.
Desire 4: Pursue an academic career.
Desire 5 (Int.): Make sure to graduate with honours.
Desire 6 (Int.): Study abroad.
- **Intentions** are the most important thing.
- **Beliefs** and **intentions** generate **desires**.
- **Desires** can be inconsistent with each other.
- **Intentions** are recomputed based on the current intentions, **desires** and beliefs.
- **Intentions should persist, normally.**
- **Beliefs are constantly updated and thus generate new desires.**
- From time to time intentions need to be re-examined.
Agent Control Loop Version 3

1.  
2.  $B := B_0;$
3.  $I := I_0;$
4.  while true do
5.       get next percept $\rho$;
6.       $B := brf(B, \rho);$  
7.       $D := options(B, I);$  
8.       $I := filter(B, D, I);$  
9.       $\pi := plan(B, I);$  
10.  execute($\pi$)
11.  end while
Deliberation has been split into two components:

1. Generate options (desires).
2. Filter the right intentions.

\((B, D, I)\) where \(B \subseteq \text{Bel}, D \subseteq \text{Des}, I \subseteq \text{Int}\)

\(I\) can be represented as a **stack** (priorities are available)
An agent has commitments both to

end: the wishes to bring about,

means: the mechanism to achieve a certain state of affairs.

⇝ Means-end reasoning.

What is wrong with our current control loop?

It is overcommitted to both means and end.

No way to replan if something goes wrong.
Agent Control Loop Version 4
1.
2. \( B := B_0; \)
3. \( I := I_0; \)
4. while true do
5. \( \text{get next percept } \rho; \)
6. \( B := \text{brf}(B, \rho); \)
7. \( D := \text{options}(B, I); \)
8. \( I := \text{filter}(B, D, I); \)
9. \( \pi := \text{plan}(B, I); \)
10. while not empty(\( \pi \)) do
11. \( \alpha := \text{hd}(\pi); \)
12. \( \text{execute}(\alpha); \)
13. \( \pi := \text{tail}(\pi); \)
14. \( \text{get next percept } \rho; \)
15. \( B := \text{brf}(B, \rho); \)
16. if not sound(\( \pi, I, B \)) then
17. \( \pi := \text{plan}(B, I) \)
18. end-if
19. end-while
20. end-while
2 Basic Notions

2.2 BDI-Architecture

- goal/intention/task
- state of environment
- possible actions

planner

plan to achieve goal
What is a plan, a planning algorithm?

**Definition 2.2 (Plan)**

A plan $\pi$ is a list of primitive actions. They lead, by applying them successively, from the *initial state* to the *goal state*.

Still overcommitted to intentions!
Agent Control Loop Version 5

1. \( B := B_0 \);
2. \( I := I_0 \);
3. while true do
4.     get next percept \( \rho \);
5.     \( B := brf(B, \rho) \);
6.     \( D := options(B, I) \);
7.     \( I := filter(B, D, I) \);
8.     \( \pi := plan(B, I) \);
9.     while not empty(\( \pi \))
10. or succeeded(\( I, B \))
11. or impossible(\( I, B \)) do
12.     \( \alpha := hd(\pi) \);
13.     execute(\( \alpha \));
14.     \( \pi := tail(\pi) \);
15.     get next percept \( \rho \);
16.     \( B := brf(B, \rho) \);
17.     if not sound(\( \pi, I, B \)) then
18.         \( \pi := plan(B, I) \)
19.     end-if
20.     end-while
21. end-while
Still limited in the way the agent can reconsider its intentions.
Agent Control Loop Version 6

1. \[ B := B_0; \]
2. \[ I := I_0; \]
3. while true do
   4.       get next percept \( \rho \);
   5.       \[ B := brf(B, \rho); \]
   6.       \[ D := \text{options}(B, I); \]
   7.       \[ I := \text{filter}(B, D, I); \]
   8.       \[ \pi := \text{plan}(B, I); \]
   9.       while not (empty(\( \pi \))
                 or succeeded(I, B)
                 or impossible(I, B)) do
      10.          \( \alpha := \text{hd}(\pi); \)
      11.          execute(\( \alpha \));
      12.          \[ \pi := \text{tail}(\pi); \]
      13.          get next percept \( \rho \);
      14.          \[ B := brf(B, \rho); \]
      15.          \[ D := \text{options}(B, I); \]
      16.          \[ I := \text{filter}(B, D, I); \]
      17.          if not sound(\( \pi, I, B \)) then
      18.                 \[ \pi := \text{plan}(B, I) \]
      19.          end-if
      20.       end-while
      21.       end-while
But reconsidering intentions is **costly**.

**Pro-active vs. reactive**

**Extreme**: stubborn agents, unsure agents.

**What is better?** Depends on the environment.

Let $\gamma$ the **rate of world change**.

1. $\gamma$ small: stubbornness pays off.
2. $\gamma$ big: unsureness pays off.

**What to do?**

**Meta-level control**
Agent Control Loop Version 7

1. \( B := B_0 \);
2. \( I := I_0 \);
3. while true do
4.     get next percept \( \rho \);
5.     \( B := brf(B, \rho) \);
6.     \( D := \text{options}(B, I) \);
7.     \( I := \text{filter}(B, D, I) \);
8.     \( \pi := \text{plan}(B, I) \);
9.     while not (empty(\( \pi \))
       or succeeded(\( I, B \))
       or impossible(\( I, B \))) do
10. \( \alpha := \text{hd}(\( \pi \)) \);
11. execute(\( \alpha \)));
12. \( \pi := \text{tail}(\( \pi \)) \);
13. get next percept \( \rho \);
14. \( B := brf(B, \rho) \);
15. if reconsider(\( I, B \)) then
16.     \( D := \text{options}(B, I) \);
17.     \( I := \text{filter}(B, D, I) \);
18. end-if
19. if not sound(\( \pi, I, B \)) then
20.     \( \pi := \text{plan}(B, I) \)
21. end-if
22. end-while
23. end-while
2.3 PROLOG
Prolog

Prolog = programmation en logique

is a logic programming language that is based on Horn clauses and resolution. We also use negation as failure to deal with incomplete information.

Programming constructs of Prolog that are important for our course are:

- terms,
- facts (also called atoms), and
- rules.

Other important notions are queries, and predefined constructs like arithmetical expressions, and lists.
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** $f(t_1, t_2, \ldots, t_n)$ composed using constants, variables and functors:

  $s(0), s(s(0)), f(c_1, f(c_1, f(s(0), c_2))),$

  first_name_of(einstein), father_of(X), ...  

- **Ground Terms** are terms without variables. They are also called **fully instantiated**.
Facts (Atoms)

They express that a relation holds between objects: They can be true or not.

\[
\begin{align*}
\text{mother}(\text{sarah}, \text{isaac}) . \\
\text{mother}(\text{lea}, \text{dina}) . \\
\text{mother}(\text{sarah}, \text{ismael}) . \\
\text{male}(\text{esau}) . \\
\text{female}(\text{dina}) . \\
\text{father}(\text{terach}, \text{abraham}) . \\
\text{father}(\text{abraham}, \text{isaac}) . \\
\text{father}(\text{abraham}, \text{ismael}) . \\
\text{father}(\text{isaac}, \text{esau}) . \\
\text{father}(\text{isaac}, \text{jakob}) . \\
\text{father}(\text{jakob}, \text{dina}) .
\end{align*}
\]

\text{father} is also called a binary predicate. Similarly, facts are sometimes called predicates.
Facts (Atoms) (2)

- The Prolog term `father(Y, father(Y, X))` is meaningless and not well-formed.
- One could consider `plus(X, Y)` as a binary function.
- Then `plus(1, plus(1, 1))` would make sense (and evaluate to something like 3, if this were available in the language).
Facts (Atoms) (3)

- A belief base always consists of facts.
- If a belief base does not contain a particular atom, say \( \text{father}(\text{isaac}, \text{terach}) \), then we can also say that “not \( \text{father}(\text{isaac}, \text{terach}) \)” is true.
- Such negated facts are also called negated atoms. We use the notion literal, to denote an atom or its negation.
- Therefore a belief base is always consistent: It can not contain any contradictory information.
Rules, Clauses (1)

To define **new predicates:**

\[
\begin{align*}
\text{son}(X,Y) & : \leftarrow \text{father}(Y,X), \text{male}(X). \\
\text{daughter}(X,Y) & : \leftarrow \text{parent}(Y,X), \text{female}(X). \\
\text{grandfather}(X,Y) & : \leftarrow \text{father}(X,Z), \text{parent}(Z,Y). \\
\text{grandmother}(X,Y) & : \leftarrow \text{mother}(X,Z), \text{parent}(Z,Y). \\
\text{parent}(X,Y) & : \leftarrow \text{father}(X,Y). \\
\text{parent}(X,Y) & : \leftarrow \text{mother}(X,Y). \\
\text{Sibling}(X,Y) & : \leftarrow \text{parent}(Z,X), \text{parent}(Z,Y).
\end{align*}
\]
Rules, Clauses (2)

They are also used to state important properties and relations between predicates:

\[
\begin{align*}
\text{male}(Y) & : - \text{father}(Y, X). \\
\text{female}(Y) & : - \text{mother}(Y, X).
\end{align*}
\]
Queries

Given a set of rules and some facts (in a belief base), it is interesting to know whether something can be deduced from that (see the Wumpus example). We can ask queries: They can be true, they can fail, or, if they contain variables, they can result in an instantiation of the variables.

\[
\begin{align*}
\text{son}(\text{isaac, abraham})? & \quad \text{true} \\
\text{plus}(1, 1, 2)? & \quad \text{true} \\
\text{daughter}(X, \text{lea})? & \quad \text{true, } X=\text{dina} \\
\text{grandmother}(X, \text{esau})? & \quad \text{true, } X=\text{sarah} \\
\text{siblings}(\text{esau, jakob})? & \quad \text{true} \\
\text{mother}(\text{terach, } Y)? & \quad \text{false?} \\
\text{plus}(1, 1, Y)? & \quad \text{true, } Y=2 \\
\text{plus}(X, X, Y)? & \quad \text{true, } X=0, Y=0
\end{align*}
\]
How to interpret the rules?

Example 2.3 (SLD-Resolution)

Let a program consist of the following rules

(1) \( p(X, Z) : \neg q(X, Y), p(Y, Z) \)
(2) \( p(X, X) \).
(3) \( q(a, b) \).

The query \( Q \) we are interested in is “\( p(X, b) \)”. I.e. we are looking for all instances (terms) \( t \) for \( X \) such that \( p(t, b) \) follows from the program.
2.3 PROLOG

\[ p(X, b) \]

\[ q(X, Y), p(Y, b) \]

\[ X/b \]

```
Success
```

```
Failure
```

\[ X/a \]

```
Success
```

```
```

\[ p(b, b) \]

\[ q(b, u), p(u, b) \]

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```
Lists

We often use **lists** and consider \([\cdot]\) as a **function symbol**, written in **infix notation**.

<table>
<thead>
<tr>
<th>List</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>([\ ])</td>
<td>empty list</td>
</tr>
<tr>
<td>([a])</td>
<td>list with one element</td>
</tr>
<tr>
<td>([a, b])</td>
<td>list with two elements</td>
</tr>
<tr>
<td>([a, b, c])</td>
<td>list with three elements</td>
</tr>
<tr>
<td>([a, [b, c]])</td>
<td>list of lists</td>
</tr>
</tbody>
</table>
# Predicates for lists

We assume we have a list of built-in predicates:

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>member(a, [a, b, c])</code></td>
<td>membership</td>
</tr>
<tr>
<td><code>member(X, [a, b, c])</code></td>
<td>membership</td>
</tr>
<tr>
<td><code>prefix([a, b], [a, b, c])</code></td>
<td>prefix</td>
</tr>
<tr>
<td><code>suffix([b, c], [a, b, c])</code></td>
<td>suffix</td>
</tr>
<tr>
<td><code>sublist([b], [a, b, c])</code></td>
<td>sublist</td>
</tr>
<tr>
<td><code>append([a, b], [c, d], [a, b, c, d])</code></td>
<td>appending two lists</td>
</tr>
</tbody>
</table>
Append

Suppose for a moment we do not have the `append` predicate available.

How can we define it using rules?

\[
\text{append}([], X, X) : - \\
\text{append}([X|Y], Z, [X|T]) : - \text{append}(Y, Z, T)
\]
Order of atoms

How can we define the reverse of a list?

Example 2.4 (Termination depends on Order)

Consider the following two programs:

1. \( \text{reverse}([X|Y], Z) : – \text{append}(U, [X], Z), \text{reverse}(Y, U) \)

2. \( \text{reverse}([X|Y], Z) : – \text{reverse}(Y, U), \text{append}(U, [X], Z) \)

together with the above definition for \( \text{append} \) and the query “\( Q : \text{reverse}([a|X], [b, c, d, b]) \)”. 
Order of atoms (cont.)

- The first program (1) leads to:
  \[ Q^1 : \text{append}(U, [a], [b, c, d, b]), \text{reverse}(X, U) \]
- The second program (2) leads to:
  \[ Q^2 : \text{reverse}(X, U), \text{append}(U, [a], [b, c, d, b]) \]
- We get different results using a naive execution!
  The first fails (no unification), the second does not terminate.
- This problem has been solved: Just do not care too much about the ordering!
not: negation-as-failure (1)

not has a very special meaning.

\[
\begin{align*}
reachable(X) & : - \ edge(X, Y), \ reachable(Y). \\
edge(a, b). \\
edge(b, a). \\
edge(c, d). \\
out\_of\_reach(X) & : - \ \textbf{not} \ reachable(X).
\end{align*}
\]

What about the query \textit{out\_of\_reach(c)}?
not: negation-as-failure (2)

Remember $female(X)$, $male(X)$. These predicates exclude each other. **How to express this with rules?**

$$
female(X) : - \ not \ male(X).
\$

$$
\$male(X) : - \ not \ female(X).
\$

This ensures that we always have $male(c)$ or $female(c)$ in a belief base, but never both (unless explicitly stated).
System Functions

These (and other) functions are pre-defined:

<table>
<thead>
<tr>
<th>Function</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sum:</td>
<td>$1 + 1$</td>
</tr>
<tr>
<td>Quotient:</td>
<td>$5 / 8$</td>
</tr>
<tr>
<td>Minus:</td>
<td>$-34$</td>
</tr>
<tr>
<td>Square root:</td>
<td>$\sqrt{16}$</td>
</tr>
<tr>
<td>Integer:</td>
<td>$\text{int}(2.1)$</td>
</tr>
<tr>
<td>Time:</td>
<td>$\text{cputime}$</td>
</tr>
<tr>
<td>Floor:</td>
<td>$\text{floor}(2.5)$</td>
</tr>
<tr>
<td>Subtract:</td>
<td>$2 - 3$</td>
</tr>
<tr>
<td>Multiply:</td>
<td>$13 \times 21$</td>
</tr>
<tr>
<td>Absolute:</td>
<td>$\text{abs}(2)$</td>
</tr>
<tr>
<td>Pi:</td>
<td>$\pi$</td>
</tr>
<tr>
<td>Random:</td>
<td>$\text{random}(16)$</td>
</tr>
<tr>
<td>Ceiling:</td>
<td>$\text{ceil}(2.5)$</td>
</tr>
<tr>
<td>Assign:</td>
<td>$\text{is}(X, 3)$</td>
</tr>
</tbody>
</table>
Downloads

You can download SWI-Prolog here:

http://www.swi-prolog.org/

And you can download the relatives-example from our homepage.
3. Some Scenarios

- Wumpus
- Agent Contest
Content of this chapter:

We present two interesting scenarios and our agent contest.

- **Wumpus**: a simple yet difficult to solve deterministic (but incomplete) environment.

- **Agent Contest**: where agents need to **collaborate** together to achieve a goal in an indeterministic environment.
3.1 Wumpus
3 Some Scenarios

3.1 Wumpus

```
1 2 3 4
1
2
3
4
START
Gold
Stench
Breeze
PIT
```

![Diagram of the Wumpus world]

- **Stench**
- **Breeze**
- **PIT**

---

Prof. Dr. Jürgen Dix - Department of Informatics, TUC
### Some Scenarios

#### 3.1 Wumpus

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>G</th>
<th>P</th>
<th>S</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,4</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
</tr>
<tr>
<td>1,3</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
</tr>
<tr>
<td>1,2</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
</tr>
<tr>
<td>1,1</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
</tr>
<tr>
<td>2,4</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
</tr>
<tr>
<td>3,4</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
</tr>
<tr>
<td>4,4</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
</tr>
</tbody>
</table>

**Symbols:**
- **A**: Agent
- **B**: Breeze
- **G**: Glitter, Gold
- **OK**: Safe square
- **P**: Pit
- **S**: Stench
- **V**: Visited
- **W**: Wumpus

### (a)

<table>
<thead>
<tr>
<th></th>
<th>1,1</th>
<th>2,1</th>
<th>3,1</th>
<th>4,1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,2</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
</tr>
<tr>
<td>1,3</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
</tr>
<tr>
<td>1,4</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
</tr>
</tbody>
</table>

### (b)

<table>
<thead>
<tr>
<th></th>
<th>1,1</th>
<th>2,1</th>
<th>3,1</th>
<th>4,1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,2</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
</tr>
<tr>
<td>1,3</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
</tr>
<tr>
<td>1,4</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
</tr>
</tbody>
</table>
3 Some Scenarios

3.1 Wumpus

A = Agent
B = Breeze
G = Glitter, Gold
OK = Safe square
P = Pit
S = Stench
V = Visited
W = Wumpus

OK = Safe square

V = Visited

(a)

(b)
Definition of suitable predicates

\[ S(i, j) \quad \text{cell} \ (i, j) \text{ stenches} \]
\[ B(i, j) \quad \text{cell} \ (i, j) \text{ breezes} \]
\[ Gl(i, j) \quad \text{cell} \ (i, j) \text{ glitters} \]
\[ Pit(i, j) \quad \text{cell} \ (i, j) \text{ is a pit} \]
\[ W(i, j) \quad \text{cell} \ (i, j) \text{ contains a Wumpus} \]

The first three predicates correspond to percepts of the agent.

The last two predicates can be determined based on the observations of the agent and the path it has taken.
General background knowledge

\[ \neg S(1, 1) \implies (\neg W(1, 1) \land \neg W(1, 2) \land \neg W(2, 1)) \]

\[ \neg S(2, 1) \implies (\neg W(1, 1) \land \neg W(2, 1) \land \neg W(2, 2) \land \neg W(3, 1)) \]

\[ \neg S(1, 2) \implies (\neg W(1, 1) \land \neg W(1, 2) \land \neg W(2, 2) \land \neg W(1, 3)) \]

\[ S(1, 2) \implies (W(1, 3) \lor W(1, 2) \lor W(2, 2) \lor W(1, 1)) \]

+ many more!!

These have to be rewritten in the form of rules \( a : \neg b \).
Belief base in initial state:

\[ \neg W(1, 1), \neg S(1, 1), \neg Pit(1, 1), \neg B(1, 1) \]

Belief base after first move:

\[ \neg W(1, 1), \neg S(1, 1), \neg Pit(1, 1), \neg B(1, 1), \neg S(2, 1), B(2, 1) \]

Belief base after second move:

\[ \neg W(1, 1), \neg S(1, 1), \neg Pit(1, 1), \neg B(1, 1), \neg S(2, 1), B(2, 1) \]
Belief base after the 3rd move:

\[ \neg W(1,1), \neg S(1,1), \neg Pit(1,1), \neg B(1,1), \neg S(2,1), B(2,1), \\
S(1,2), B(2,1), \neg B(1,2) \]

Question:
Can we deduce that the wumpus is located at (1,3)?

Answer:
Yes. This can be done automatically using built-in features of the programming language: By querying \( W(1,3) \) against the belief base.
3.2 Agent Contest
Scenario: Gold Miners

Task: Implement a team of agents that collects more gold than the opponent.

Aim: Agents should cooperate and coordinate their actions. Agents can take on roles and split into subgroups to solve the overall task more efficiently.

Emerging behaviour instead of a hard-wired solution.

Environment: Can be quite indeterministic: percepts can be blurred, actions could fail with certain probability, ...
3 Some Scenarios

3.2 Agent Contest Environment

Figure: Elements in the environment

- Agents
- Gold
- Obstacles
- Depot
Figure: Gold Miners 2006: CLIMABot (blue) vs. brazil (red)
Details

**Discrete Simulation:** in each step do
- send perceptions to agents
- wait for agents’ actions or timeout
- let agents act

**Tournament Structure:**
- maximum step duration around 4 seconds
- approx. 1000 steps per simulation
- 3 simulations = 1 match
- each team plays against all others, 1 match per pair
Technical details

- Grid size: $30 \times 30$
- Perception failure: 1%
- Action failure: 2%
- Occupying the depot leads to teleporting the agent
Lab exercise

We will implement agent teams in the lab exercises and let the teams compete against each other. The better one wins!
Scenario: Cows and Cowboys

Task: Implement a team of agents that collects more cows than the opponent.

Aim: Agents have to cooperate and coordinate their actions. Agents can take on roles and split into subgroups to solve the overall task more efficiently.

Emerging behaviour instead of a hard-wired solution.

Environment: Can be quite indeterministic: behaviour of cows, percepts can be blurred, actions could fail with certain probability, ...
Environment

- Cows
- Cowboys
- Corrals
- Obstacles
What is the optimal solution?

We do not know!
Details

**Discrete Simulation:** in each step do
- send perceptions to agents
- wait for agents’ actions or timeout
- let agents act and move cows

**Tournament Structure:**
- maximum step duration around 4 seconds
- approx. 1000 steps per simulation
- 3 simulations = 1 match
- each team plays against all others, 1 match per pair
Agents

- fixed visibility range (square)
- actions: move to one of eight directions
Cows

- visibility range (square)
- afraid of: agents, obstacles
- feel good: near other cows and empty spaces
- actions: move to one of eight directions
- slower than agents
Map: Razoredge
Map: Cowskullmountain
Agent Contest: Chasing Cows
4. Jason

- Origins and Fundamentals
- Features
- Multi-Agent System definition
- Tools
4.1 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 behaviour
- Social ability
- Combination of theoretical and practical reasoning
- E.g. of the best known and publicly available languages

- Jadex (Pokahr, Braubach)  http://jadex.sf.net
- 2APL (Dastani, Meyer)  http://www.cs.uu.nl/2apl
- Jason (Bordini, Hübner)  http://jason.sf.net
AgentSpeak

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

Syntax

Beliefs are represented by annotated literals of first order logic

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

Example 4.1 (Belief base of agent Tom)

\[
\begin{align*}
\text{red(box1)}[\text{source(percept)}]. \\
\text{friend(bob,alice)}[\text{source(bob)}]. \\
\text{lier(alice)}[\text{source(self)},\text{source(bob)}]. \\
\sim\text{lier(bob)}[\text{source(self)}].
\end{align*}
\]
Changes in the belief base I

By perception

beliefs annotated with $\text{source}(\text{percept})$ 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
```
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 4.2 (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

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

```plaintext
.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

```c
.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)
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
**Plans**

### Boolean operators
- \& (and)
- | (or)
- **not** (not)
- = (unification)
- >, >= (relational)
- <, <= (relational)
- == (equals)
- \= (different)

### Arithmetic operators
- + (sum)
- - (subtraction)
- * (multiply)
- / (divide)
- div (divide – integer)
- mod (remainder)
- ** (power)
Plans — Operators for plan’s body

A plan’s body may contain:

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

Example 4.3 (Plan’s body)

\[
\begin{align*}
+\text{beer} : & \quad \text{now}(H) \land \text{time}_{\text{to}}_{\text{leave}}(T) \land H \geq T \\
<- !g1; & \quad \text{// new sub-goal} \\
!!g2; & \quad \text{// new goal} \\
+b1(T-H); & \quad \text{// add new self belief} \\
+-b2(T*H); & \quad \text{// update belief} \\
?b(X); & \quad \text{// new sub-goal} \\
X > 10; & \quad \text{// constraint to continue the plan} \\
\text{close}(\text{door}). & \quad \text{// external action}
\end{align*}
\]
Plans — example

+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).
Jason basic reasoning cycle

- perceive the environment and update belief base
- process new messages
- select event
- select **relevant** plans
- select **applicable** plans
- create/update intention
- select intention to execute
Consider a very simple robot with two goals:

- when a piece of gold is seen, go to it
- when battery is low, charge

Example 4.4 (Java code – go to gold)

```java
public class Robot extends Thread {
    boolean seeGold, lowBattery;

    public void run() {
        while (true) {
            while (! seeGold) {
            }
            while (seeGold) {
                a = selectDirection();
                doAction(go(a));
            } }
    }
```
Example 4.5 (Java code – charge battery)

```java
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)
Example 4.6 (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
4.2 Features
Strong negation

Example 4.7

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

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

Example 4.8

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

Example 4.9 (An agent blindly committed to $g$)

$+!g : g.$

$+!g : \ldots \leftarrow \ldots \ ?g.$

$-!g : \text{true} \leftarrow !g.$
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:

  ```java
  libname.action_name(...)
  ```
Standard Internal actions

- Standard (pre-defined) internal actions have an empty library name
  - .print\(\text{term}_1, \text{term}_2, \ldots\)
  - .union\(\text{list}_1, \text{list}_2, \text{list}_3\)
  - .my_name\(\text{var}\)
  - .send\(\text{ag}, \text{perf}, \text{literal}\)
  - .intend\(\text{literal}\)
  - .drop_intention\(\text{literal}\)

- Many others available for: printing, sorting, list/string operations, manipulating the beliefs/annotations/plan library, creating agents, waiting/generating events, etc.
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, ....)
4.3 Multi-Agent System definition
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

- There will normally 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.
Interaction with the environment simulator

Environment Simulator  | Agent Architecture  | Reasoner

getPercepts → Agent Architecture
executeAction → Agent Architecture
change percepts → Environment Simulator

Agent Architecture → perceive
Agent Architecture → act
Reasoner → perceive
Reasoner → act
Example of an Environment Class

```java
import jason.*;
import ...;
public class robotEnv extends Environment {
    ....
    public robotEnv() {
        Literal gp =
            Literal.parseLiteral("green_patch(souffle)");
        addPercept(gp);
    }

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

- Simple way of defining a multi-agent system

Example 4.10 (MAS that uses 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"
}

Prof. Dr. Jürgen Dix · Department of Informatics, TUC
MAS configuration language II

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

Example 4.11 (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",
        ...
    )
}
```
4.4 Tools
jEdit plugin
Eclipse plugin

```java
/*
 * Jason Project
 * -- create on December 16, 2007
 */

MAS jp2 {
  infrastructure: Centralised
  environment: a.E
  agents:
    sample;
  aslSourcePath: "src/asl";
}
```
Mind inspector

Inspection of agent r1 (cycle #12)

- **Beliefs**
  - pos(back,3,0)_{source(agent)}
  - pos(r1,3,0)_{source(percept)}
  - pos(r2,3,3)_{source(percept)}
  - garbage(r1)_{source(percept)}

- **Events**
  - | Sel | Trigger | Intention |
  - |-----|---------|-----------|
  - | X   | +ensure_pick(garb) | 4 |

- **Options**

- **Intentions**
  - | Sel | Id | Pen | Intended Means Stack (show details) |
  - |-----|----|-----|-----------------------------------|
  - | X   | 4  |     | +ensure_pick(S) { S = garb }     |
  - |     |    |     | +take(S,L) { S = garb, L = r2 } |
  - |     |    |     | +carry_to(R) { R = r2, Y = 0, X = 3 } |
  - |     |    |     | +garbage(r1)_{source(percept)} |

- **Actions**
  - | Pend | Feed | Sel | Term | Result | Intention |
  - |------|------|-----|------|--------|-----------|
  - | X    | X    | pick(garb) | false | 4      |
More information


- Bordini, R. H., Hübner, J. F., and Wooldrige, M.
  Programming Multi-Agent Systems in AgentSpeak using *Jason*