Multiagent Systems I

Prof. Dr. Jürgen Dix

Department of Informatics
Clausthal University of Technology
SS 2012

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. Only the first two weeks are in class, the rest are labs where students are programming a team to compete in our newest agent contest scenario. Students are grouped into teams and implement agent teams for solving a task on our agent contest platform. We consider BDI as a basic framework for developing agents using JAVA.

My thanks go to Tristan, Michael, Federico and our students who prepared the lab work and also some of the slides of this course.

References


1. Week: 1. Introduction,
   2.1 Reactive Agents
2. Week: 2.2 BDI, 3. Searching
3. Week: 4. Agent Contest Scenario
4.-15. Week: Labs.

1. Introduction

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


Example 1.2 (RoboCup)

Figure: 2D-Simulation league: RoboCup 2007 Final
1 Introduction

1.1 Why Agents?

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
1 Introduction

1.1 Why Agents?

Example 1.7 (RoboCup)

Figure: Humanoid league

Example 1.8 (RoboCup)

Figure: Rescue league

Example 1.9 (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

Figure: Grand Challenge 2004
Example 1.10 (Grand Challenge 2005)
Second try: **Big Success**: 
**Stanley** (Sebastian Thrun) won in 2005.

![VW Touareg coached by Stanford University](image)

- 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 in 2007.

![Urban Challenge 2007](image)

- 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

Example 1.13 (Agent Contest: Chasing Cows)
Second try: **Push cows in a corral.**
- http://multiagentcontest.org

Figure: Chasing Cows 2008
1 Introduction

1.1 Why Agents?

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.

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

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.

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. 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 Introduction

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:

set of percept sequences $\mapsto$ 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>
<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 score on test</td>
<td>Set of students, testing agency</td>
<td>Display exercises, suggestions, corrections</td>
<td>Keyboard entry</td>
</tr>
</tbody>
</table>

Table: Examples of agents types and their PEAS descriptions.

**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/Continuous: Density of observations and actions. Relevant: Level of granularity.
### Environment Accessible Deterministic Episodic Static Discrete

<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>No</td>
<td>No</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>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</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>Yes</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 1.18 (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.)

In Theory Everything is Trivial

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

A Simple Agent Program

```java
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
```

```java
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
```
1 Introduction

1.2 Intelligent Agents

Example 1.20 (Agent: Taxi Driver)

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

```python
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.

**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

$$\text{action} : S \rightarrow A.$$ 

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 1.22 (Run)**

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 1.23 (Environment, 2. version)**

An environment $Env$ is a triple $(S, s_0, \tau)$ consisting of

1. the set $S$ of states,
2. the initial state $s_0 \in S,$
3. a function $\tau : R^{act} \rightarrow 2^S$, 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

$$\text{action} : R^{state} \rightarrow A.$$ 

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

**Important:**

An agent system is then a pair $a = \langle \text{action}, Env \rangle$ consisting of an agent and an environment. We denote by $R(a, Env)$ the set of runs of agent $a$ in environment $Env.$
Definition 1.25 (Characteristic Behaviour)

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

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

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 $Env$.

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)

$A := \{a_1, a_2, ..., a_n\}$ is the set of actions.

$P := \{p_1, p_2, ..., p_m\}$ is the set of percepts.

$S := \{s_1, s_2, ..., s_l\}$ is the set of states.

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!)
Sensors don’t need to provide perfect information!

**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

\[ \text{action : } P \rightarrow A. \]

**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

**A Very Simple Reflex Agent**

```python
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 $\alpha$)

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

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

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

Definition 1.29 (Indistinguishable)

Two different states $s, s'$ are indistinguishable for an agent $\alpha$, 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 1.30 (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

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

where

\[
p_0 = \text{see}(s_0),
\]

\[
a_i = \text{action}(\langle p_0, \ldots, p_i \rangle),
\]

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

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

\[
s_0 \rightarrow a_0 \ s_1 \rightarrow a_1 \ldots s_n \rightarrow a_n \ldots
\]
Instead of using the whole history, resp. $P^\ast$, one can also use internal states $I := \{i_1, i_2, \ldots, i_n, i_{n+1}, \ldots\}$.

**Definition 1.31 (State-based Agent $a_{\text{state}}$)**

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

- $\text{see} : S \rightarrow P$,
- $\text{next} : I \times P \rightarrow I$.

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

**Definition 1.32 (Characteristic Behaviour)**

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

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

with

- $p_0 = \text{see}(s_0)$,
- $p_i = \text{see}(s_i)$, 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 $r : s_0 \Rightarrow a_0 s_1 \Rightarrow a_1 \cdots$ 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 1.33 (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 1.34 (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.

2. Basic Architectures

2. Basic Architectures
- Reactive Agents
- BDI-Agents

Content of this Chapter:
We are presenting two very basic architectures: a simple subsumption architecture, and an important paradigm of agent programming: The BDI-framework. While this is a very general framework, several programming languages can be seen as implementations of BDI.

1. We present a simple model for a subsumption architecture.
2. We discuss the agent control loop of BDI through several stages.
3. We introduce means-end reasoning.
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.
- Many behaviours can be applied **concurrently**. How to select between them? Implementation through Subsumption-Hierarchies, Layers. Upper layers represent abstract behaviour.

**Formal Model:**

- **see**: Close relation between observation and action: **no transformation of the input**.
- **action**: Set of behaviours and a relation.

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

\( \langle c, a \rangle \) “fires” if see(s) \( \in c \)

\( \langle \cdot, \cdot \rangle \subseteq A_{rules} \times A_{rules} \)

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

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

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

\( b_1 \) has priority over \( b_2 \).
2 Basic Architectures

2.1 Reactive Agents

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.

Gradient Field

The spaceship sends off radio signals: gradient field.

Assumptions

Under which assumptions (on the distribution of the gold) does this work perfectly?

Low Level Behaviour:

1. If detect an obstacle then change direction.

2. Layer: If Samples on board and at base then drop off.

(3) If Samples on board and not at base then follow gradient field.

3. Layer: If Samples found then pick them up.

4. Layer: If true then take a random walk.

With the following ordering

\[(1) \prec (2) \prec (3) \prec (4) \prec (5)\]
Vehicles can communicate indirectly with each other:

- they put off, and
- pick up radioactive samples that can be sensed.

**Pro:** Simple, economic, efficient, robust, elegant.

**Contra:**

- Without knowledge about the environment agents need to know about their 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 ≤ 10 behaviours are doable. But the more layers the more complicated to understand what is going on.
What is BDI?

BDI is based on the assumption that the mind, that is the mental state, of agents consists of:

- **beliefs**: what the agent believes to be true about the world (information).
- **desires**: which state(s) of the world the agents wants to establish (motivation).
- **intentions**: what the agent actually intends to do and how to do it (deliberation).

The world of an agent is the other agents, the environment, and the agent itself.

Where does it come from?

BDI builds on three subfields of artificial intelligence:

- rational agents,
- planning, and
- decision theory.

What is it good for?

BDI allows for

- means-end reasoning,
- weighing of competing possibilities,
- the interaction between these two forms of reasoning, and addresses the problem of resource-boundedness.

Means-end reasoning

- comes from the subfield of AI that deals with planning

- **Given**: an initial state, a set of goal states (ends), and a description of actions (means or capabilities)

- **Goal**: find a sequence of actions (plan) that leads from the goal state to the final state
2 Basic Architectures

2.2 BDI-Agents

Means-end reasoning (2)

Example:

- initial state: I am at home, I have a picture, I have nails, I have no frame and no tools.
- goal state: the picture is framed and hangs on the wall
- plan: 1. go to DIY store, 2. acquire a frame and a hammer, 3. go home, 4. frame the picture, 5. use hammer and nails to hang the picture on the wall

Weighing of competing possibilities

- comes from decision theory
- competing possibilities are taken as given
- weigh the possibilities and decide for one of them, that is select an option based on the agent’s utility function which takes into account beliefs (what the agent knows) and desires (what the agent wants)

Weighing of competing possibilities (2)

Example:

- desire: have a meal
- possibility 1: go to Mensa
- possibility 2: go to a fancy restaurant
- beliefs: I am low on funds
- decision: go to Mensa

Resource-Boundedness

- Agents are resource bounded, that is
- they are unable to perform arbitrarily large computations in the available time.
Necessity of BDI

Important characteristics of real-time applications:

1. The environment is nondeterministic, i.e., in each state of the environment can evolve in several ways.
2. The system is nondeterministic, i.e., in each state there are potentially several different actions to perform.
3. The system can have several different objectives at the same time.
4. The actions/procedures that achieve the objectives best are dependent on the state of the environment and are independent of the internal state of the system.
5. The environment can only be sensed locally.
6. The rate at which computations and actions can be carried out is within reasonable bound to the rate at which the environment evolves.

Necessity of BDI II

Nondeterminism of the environment (1) and of the system (2) imply a formal model:

Branching Tree Structure

- Each node is a certain state of the world,
- each transition represents a primitive action made by the system, a primitive event occurring in the environment, or both,
- each branch represents an alternative execution path,
- choice nodes are manifestations of the system's nondeterminism, and
- chance nodes are manifestations of the environment's nondeterminism.

Branching Tree Structure Example

nodes transitions an exemplary path choice nodes chance nodes

Selection Function

The branching tree structure requires a selection function, that selects appropriate actions to execute from the various available options.

How should the selection function look like? What should be the right data-structures?
2 Basic Architectures

2.2 BDI-Agents

Necessity of BDI III

Characteristics 4 (best action dependent on environment-state and independent of internal system-state), 1 (environment-nondeterminism), and 5 (local sensing) imply that it is necessary that there is some component of the system that can represent the information about the state of the world.

⇝ Beliefs!

Necessity of BDI IV

Characteristics 3 (several parallel objectives) and 5 (local sensing) imply that it is necessary that the system also has information about the objectives to be accomplished.

⇝ Desires!

Necessity of BDI V

Idea: reconsider the choice of action at each step.

Dilemma: this is potentially too expensive and the chosen action might possibly be invalid when selected.

Assumption: it is possible to limit the frequency of reconsideration and achieve a balance between too much and not enough reconsideration. Remember characteristic 6 (rate of computations and actions is reasonable).

Implication: it is necessary to include a component of the system that represents the currently chosen course of action.

⇝ Intentions!

Basic Data-Structures (Practice)

- **Set of beliefs**: Usually stored in a belief-base.
  
  **Example**:
  - I am a student of computer-science.
  - I am in my third semester.

- **Set of goals**: Usually stored in a goal-base.
  
  **Example**:
  - I want to graduate in computer science.

- **Set of plans**: Recipes of how to reach the goals. Usually somehow structured, e.g. nested actions, and stored in a plan-base.
Example:

1. Become a bachelor of science.
   - Attend some lectures.
   - Succeed in a lot of exams.
   - Earn a living.

2. Become a master of science.
   - Attend more lectures.
   - Succeed in even more exams.
   - Earn a living.

Usually the mental attitudes are based on a knowledge representation language, e.g. Prolog.

- **Beliefs**: `studies(me, computer_science). semester(me, 3).`
- **Goals**: `graduate(me, computer_science).`
- **Plan**: `[attend(info1), attend(l_algebra), ...]`

Agent Control Loop v1

```
while true do
  observe the world;
  update the internal world model;
  deliberate about what intention to achieve next;
  use means-ends reasoning to get a plan for the intention;
  execute the plan;
end while
```

⇝ very high-level

We have to answer three questions

1. **Deliberation**: How to deliberate? That is carefully considering and weighing options.
2. **Planning**: Once committed to something, how to reach the goal?
3. **Replanning**: What if during execution of the plan, things are running out of control and the original plan fails?
Agent Control Loop v2

Set<Belief> beliefs = initBeliefBase();
while (true) {
    Percept percept = getNextPercept();
    beliefs = beliefRevision(beliefs, percept);
    Set<Intention> intentions = deliberation(beliefs);
    Plan plan = generatePlan(beliefs, intentions);
    execute(plan);
}

What is the problem here?

BDI-Agent Control Loop v3

Set<Belief> beliefs = initBeliefBase();
Set<Intention> intentions = initIntentionBase();
while (true) {
    Percept percept = getNextPercept();
    beliefs = beliefRevision(beliefs, percept);
    Set<Desire> desires = findOptions(beliefs, intentions);
    intentions = filter(beliefs, desires, intentions);
    Plan plan = generatePlan(beliefs, intentions);
    execute(plan);
}

The philosophy behind

- Intentions are the most important things.
- 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.

Observation

Deliberation has been split into two components:

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

Intentions can be represented as a stacks (i.e. priorities are available).
Observation

An agent has commitments both to

- **ends**: the wishes to bring about, and
- **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 ends. No way to replan if something goes wrong.

BDI-Agent Control Loop v4

```java
Set<Belief> beliefs = initBeliefBase();
Set<Intention> intentions = initIntentionBase();
while (true) {
    Percept percept = getNextPercept();
    beliefs = beliefRevision(beliefs, percept);
    Set<Desire> desires = findOptions(beliefs, intentions);
    intentions = filter(beliefs, desires, intentions);
    Plan plan = generatePlan(beliefs, intentions);
    while (!plan.isEmpty()) {
        Action head = plan.removeFirst();
        execute(head);
        percept = getNextPercept();
        beliefs = beliefRevision(beliefs, percept);
        if (!sound(plan, intentions, beliefs)) {
            plan = generatePlan(beliefs, intentions);
        }
    }
}
```

What is a plan?

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

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

plan to achieve goal

planner

What is wrong with our control loop?

It is still overcommitted to intentions.

BDI-Agent Control Loop v5

Set<Belief> beliefs = initBeliefBase();
Set<Intention> intentions = initIntentionBase();
while(true)
{
    Percept percept = getNextPercept();
    beliefs = beliefRevision(beliefs,percept);
    Set<Desire> desires = findOptions(beliefs,intentions);
    intentions = filter(beliefs,desires,intentions);
    Plan plan = generatePlan(beliefs,intentions);
    while(!(plan.isEmpty() || succeeded(intentions,beliefs) ||
           impossible(intentions,beliefs)))
    {
        Action head = plan.removeFirst();
        execute(head);
        percept = getNextPercept();
        beliefs = beliefRevision(beliefs,percept);
        if (sound(plan,intentions,beliefs))
        {
            plan = generatePlan(beliefs,intentions);
        }
    }
}

What is wrong with our control loop?

It is still limited in the way the agent can reconsider its intentions.
2 Basic Architectures

2.2 BDI-Agents

BDI-Agent Control Loop v6

Set Belief beliefs = initBeliefBase();
Set Intention intentions = initIntentionBase();
while (true) {
    Percept percept = getNextPercept();
    beliefs = beliefRevision(beliefs, percept);
    Set Desire desires = findOptions(beliefs, intentions);
    intentions = filter(beliefs, desires, intentions);
    Plan plan = generatePlan(beliefs, intentions);
    while (! (plan.isEmpty() || succeeded(intentions, beliefs) ||
        impossible(intentions, beliefs))) {
        Action head = plan.removeFirst();
        execute(head);
        percept = getNextPercept();
        beliefs = beliefRevision(beliefs, percept);
        desires = findOptions(beliefs, intentions);
        intentions = filter(beliefs, desires, intentions);
        if (! sound(plan, intentions, beliefs)) {
            plan = generatePlan(beliefs, intentions);
        }
    }
    // Other code...
}

Problems

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

3 Searching

3 Searching

- Problem Formulation
- Uninformed search
3 Searching

Content of this chapter:

Searching: Search Algorithms are perhaps the most basic notion of AI. Almost any problem can be formulated as a search problem.

3.1 Problem Formulation

We distinguish four types:

1. **1-state-problems**: Actions are completely described. Complete information through sensors to determine the actual state.

2. **Multiple-state-problems**: Actions are completely described, but the initial state is not certain.

3. **Contingency-problems**: Sometimes the result is not a fixed sequence, so the complete tree must be considered.

4. **Exploration-problems**: Not even the effect of each action is known. You have to search in the world instead of searching in the abstract model.

Table: The vacuum world.
Definition 3.1 (1-state-problem)

A 1-state-problem consists of:

- a set of states (incl. the initial state)
- a set of actions (operators), which – applied to a state – leads to an other state:
  \[
  \text{Operator}_i : \text{States} \rightarrow \text{States}, \quad i = 1, \ldots, n
  \]
- a set of goal states or a goal test, which – applied on a state – determines if it is a goal-state or not.
- a cost function \( g \), which assesses every path in the state space (set of reachable states) and is usually additive.

Definition 3.2 (State Space)

The state space of a problem is the set of all reachable states (from the initial state). It forms a directed graph with the states as nodes and the arcs the actions leading from one state to another.
3 Searching

3.1 Problem Formulation

Table: Belief Space for Vacuum world without sensors.

Table: Map of Romania

3.2 Uninformed search

Principle: Choose, test, expand.

Search-tree

(a) The initial state

(b) After expanding Arad

(c) After expanding Sibiu

Table: Map of Romania
**Tree Search**

Function `TREE-SEARCH(problem, strategy)` returns a solution, or failure.

1. Initialize the search tree using the initial state of `problem`.
2. Loop do
   - If there are no candidates for expansion then return failure.
   - Choose a leaf node for expansion according to `strategy`.
     - If the node contains a goal state then return the corresponding solution.
     - Else expand the node and add the resulting nodes to the search tree.

**Table: Tree Search.**

---

**Definition 3.3 (Datatype Node)**

The datatype `node` is defined by `state` (∈ `S`), `parent` (a node), `action` (also called operator) which `generated` this node, `path-costs` (the costs to reach the node) and `depth` (distance from the root).

**Important:**

The recursive dependency between node and parent is important. If the depth is left out then a special node `root` has to be introduced.

Conversely the `root` can be defined by the depth: `root` is its own parent with depth 0.

---

**Important:**

**State-space versus search-tree:**

The search-tree is **countably infinite** in contrast to the **finite** state-space.

- a node is a bookkeeping data structure with respect to the problem instance and with respect to an algorithm;
- a state is a snapshot of the world.

---

**Figure:** Illustration of a node in the 8-puzzle.
Instantiating Tree-SEARCH

Design-decision: Queue

Tree-SEARCH generates nodes. Among them are those that are-to-be-expanded later on. Rather than describing them as a set, we use a queue instead. The fringe is the set of generated nodes that have not yet been expanded.

Here are a few functions operating on queues:

- Make-Queue(Elements)
- Remove-First(Queue)
- Empty?(Queue)
- First(Queue)
- Insert(Element,Queue)
- Insert-All(Elements,Queue)

Question:
Which are interesting requirements of search-strategies?

- completeness
- time-complexity
- space complexity
- optimality (w.r.t. path-costs)

We distinguish:

Uninformed vs. informed search.

Datatype Node

Graph-Search

Table: Tree-Search

Definition 3.4 (Completeness, optimality)
A search strategy is called

- complete, if it finds a solution, provided there exists one at all.
- optimal, if whenever it produces an output, this output is an optimal solution, i.e. one with the smallest path costs among all solutions.

Is any optimal strategy also complete? Vice versa?
Breadth-first search: “nodes with the smallest depth are expanded first”,

Make-Queue : add new nodes at the end: FIFO

Complete? Yes.
Optimal? Yes, if all operators are equally expensive.

Constant branching-factor $b$: for a solution at depth $d$ we have generated\(^1\) (in the worst case)
$$b + b^2 + \ldots + b^d + (b^{d+1} - b)$$-many nodes.

Space complexity = Time Complexity

\(^1\) Note this is different from “expanded”.

---

<table>
<thead>
<tr>
<th>Depth</th>
<th>Nodes</th>
<th>Time</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1 millisecond</td>
<td>100 bytes</td>
</tr>
<tr>
<td>2</td>
<td>111</td>
<td>.1 seconds</td>
<td>11 kilobytes</td>
</tr>
<tr>
<td>4</td>
<td>11,111</td>
<td>11 seconds</td>
<td>1 megabyte</td>
</tr>
<tr>
<td>6</td>
<td>$10^6$</td>
<td>18 minutes</td>
<td>111 megabytes</td>
</tr>
<tr>
<td>8</td>
<td>$10^8$</td>
<td>31 hours</td>
<td>11 gigabytes</td>
</tr>
<tr>
<td>10</td>
<td>$10^{10}$</td>
<td>128 days</td>
<td>1 terabyte</td>
</tr>
<tr>
<td>12</td>
<td>$10^{12}$</td>
<td>35 years</td>
<td>111 terabytes</td>
</tr>
<tr>
<td>14</td>
<td>$10^{14}$</td>
<td>3500 years</td>
<td>11,111 terabytes</td>
</tr>
</tbody>
</table>

Table : Time versus Memory.

Uniform-Cost-Search: “Nodes $n$ with lowest path-costs $g(n)$ are expanded first”

Make-Queue : new nodes are compared to those in the queue according to their path costs and are inserted accordingly

Complete? Yes, if each operator increases the path-costs by a minimum of $\delta > 0$ (see below).

Worst case space/time complexity: $O(b^{1+\frac{C^*}{\delta}})$, where $C^*$ is the cost of the optimal solution and each action costs at least $\delta$
If all operators have the same costs (in particular if $g(n) = \text{depth}(n)$ holds):

**Uniform-cost search**

Uniform-cost search = **Breadth-first search**.

**Theorem 3.5 (Optimality of Uniform-cost search)**

If $\exists \delta > 0 : g(\text{succ}(n)) \geq g(n) + \delta$ then: Uniform-cost search is **optimal**.

---

**State space vs. Search tree**

**Rectangular grid:** How many different states are reachable within a path of length $d$?

![State space and search tree diagrams](image)

**Graph-Search** = **Tree-Search** + **Loop-checking**

**Table:** State space versus Search tree: exponential blow-up.

**Function**

```
graph-search(problem, fringe) returns a solution, or failure
    closed ← an empty set
    fringe ← insert(make-node(initial-state[problem]), fringe)
    loop do
        if empty?(fringe) then return failure
        node ← remove-first(fringe)
        if goal-test[problem](state[node]) then return solution(node)
        if state[node] is not in closed then
            add state[node] to closed
            fringe ← insert-all(expand(node, problem), fringe)
    end
```

**Table:** Graph Search.
Dijkstra’s Algorithm

Given
Graph, Source node in the graph.

Problem
Find the path with lowest cost between source and all other nodes.

Concept

1. Initial distance value
2. Mark all nodes as unvisited
3. Set initial node as current
4. Consider all unvisited neighbors and calculate their distance
5. After considering all neighbors of the current node, mark it as visited.
6. All nodes visited? \(\rightarrow\) finish. Otherwise: current node := unvisited node with the smallest distance

Complexity

Worst-Case Performance:
- Dijkstra’s original algorithm does not use a min-priority queue: \(O(|V|^2)\).
- With a min-priority queue implemented by a Fibonacci heap: \(O(|E| + |V|\log|V|)\).

4. MASSim & The Multi-Agent Programming Contest

- Introduction
- The Scenario
- Programming and running the agents
This chapter is about the programming part of the lecture. We
- introduce the Multi-Agent Contest 2011 scenario Agents on Mars;
- describe the structure of the agents: properties, roles, actions, percepts;
- motivate how to program and run agents.

Multi-Agent Programming Contest 2011
http://www.multiagentcontest.org/

- annual competition, started in 2005
- Aims to stimulate research in the area of MAS development and programming by:
  1. identifying key problems,
  2. collecting suitable benchmarks, and
  3. gathering test cases which require and enforce coordinated action.

- participants include researchers from around the world and developers of the most well known multi-agent programming platforms.

What is MASSim?

- a platform for testing and comparing MAS
- discrete, step-based simulations of environments
- teams of agents compete against each other
- supports different pluggable scenarios
- client-server architecture
- platform used for the Multi Agent Programming Contest
4.2 The Scenario

The 2011 Scenario: Mars

In the year 2033 mankind finally populates Mars. While in the beginning the settlers received food and water from transport ships sent from earth shortly afterwards — because of the outer space pirates — sending these ships became too dangerous and expensive. Also, there were rumors going around that somebody actually found water on Mars below the surface. Soon the settlers started to develop autonomous intelligent agents, so-called All Terrain Planetary Vehicles (ATPV), to search for water wells. The World Emperor — enervated by the pirates — decided to strengthen the search for water wells by paying money for certain achievements.

The Mars Scenario - The tasks of the agents

- find the best water wells,
- occupy the best zones of Mars,
- sabotage the rivals,
- defend from sabotages, and
- earn money through different achievements, such as:
  - inspect certain percentages of the maps,
  - build zones worth at least some value,
  - successfully attack a number of opponent agents,
  - etc.
The Mars Map

- the map is represented by a graph
- numbers on each edge indicate the cost of traversing that edge
- numbers on each node is the score that the node will earn when part of a zone
- the visualization shows the zones conquered
- agents must find out characteristics of the map by exploring and executing specific actions

Conquering zones

To dominate parts of the map, agents must stand on specific nodes of the map.

- The algorithm for determining map domination depends on the locations of all the agents.
- The objective is to isolate parts of the map from the opponent agents!
- Agents should choose zones strategically to maximize the score.
- Coordination is required!: an agent can not build a zone on its own; at least two agents are needed.

Conquering zones - The algorithm

Consist of three steps:

1. A node with agents belongs to the team that has the majority (at least half) of the agents in that node.
2. Nodes that are neighbors of occupied nodes belong to the team that controls most of those neighbors node.
3. The previous two step may draw a frontier that isolates a part of the graph from all the agents of the other teams: if that is case, the team also dominates nodes inside that frontier.

Building zones - Example

Figure: Step 1.
4 MASSim & The Multi-Agent Programming Contest

4.2 The Scenario

Building zones - Example (2)

Figure: Step 2.

Building zones - Example (3)

Figure: Step 3.

Building zones - Example (4)

Figure: Breaking a frontier.

Attributes of an agent

Agents have the following numerical attributes:

- **energy**: consumed by performing actions.
- **health**: decreased when attacked.
- **strength**: used when attacking.
- **visibility range**: how far the the agents can perceive.

These attributes can vary during the simulation.
Available actions

The following actions are available to agents, depending on the agent’s role:

- **skip**: no action is performed.
- **recharge**: part of the energy is restored.
- **goto**: move to a neighbor node.
- **attack**: decrease health of an opponent agent.
- **parry**: defend oneself against possible attacks.
- **repair**: Restore the health of another teammate.

Available actions (2)

- **probe**: Find out the value of the current node.
- **survey**: Find out costs of edges connected to current node.
- **inspect**: Find out current attributes of other agents in the range.
- **buy**: Exchange achievement points for improvements in one attribute’s (maximum) value.

Roles

The agents are heterogeneous!
Each agent assumes one of these roles:

- Explorer
- Repairer
- Saboteur
- Sentinel
- Inspector

There are two agents of each kind in every team.

The Explorer

It can: skip, goto, probe, survey, buy, recharge.

Attributes:

- Energy (maximum): 12
- Health (maximum): 4
- Strength: 0
- Visibility range: 2
### Roles: The Repairer

It can: skip, goto, parry, survey, buy, repair, recharge.

Attributes:
- Energy (maximum): 8
- Health (maximum): 6
- Strength: 0
- Visibility range: 1

---

### Roles: The Saboteur

It can: skip, goto, parry, attack, survey, buy, recharge.

Attributes:
- Energy (maximum): 7
- Health (maximum): 3
- Strength: 4
- Visibility range: 1

---

### Roles: The Sentinel

It can: skip, goto, parry, survey, buy, recharge.

Attributes:
- Energy (maximum): 10
- Health (maximum): 1
- Strength: 0
- Visibility range: 3

---

### Roles: The Inspector

It can: skip, goto, inspect, survey, buy recharge.

Attributes:
- Energy (maximum): 8
- Health (maximum): 6
- Strength: 0
- Visibility range: 1
Disabled agents

When an agent’s health reaches 0, the agent is disabled:
- it does not count for zones building.
- only goto, repair, skip, and recharge can be executed.
- the recharging rate is lower.

Disabled agents don’t count for zones! Disabling an opponent agent may help in building your own zones or destroying the opponents’.

Achievements

When a team reaches a milestone, its money (a.k.a. achievement points) is increased.

Possible achievements are:
- having zones with fixed values, e.g. 10, 20, etc.
- fixed numbers of probed vertices, e.g. 5, 10, etc.
- fixed numbers of surveyed edges, e.g. 10, 20, etc.
- fixed numbers of inspected vehicles, e.g. 5, 10, etc.
- fixed numbers of successful attacks, e.g. 5, 10, etc. or
- fixed numbers of successful parries, e.g. 5, 10. etc.

Perceptions

In every simulation step, agents perceive:
- state of the simulation: the current step,
- state of the team: the current scores and money,
- state of the agent: its internals as described previously,
- visible vertices: identifier and dominating team,
- visible edges: its vertices’ identifiers,
- visible agents: its identifier, vertex, team.

Perceptions (2)

Some elements are only perceived after an specific action. These elements are:
- probed agents: its identifier and its value,
- surveyed edges: its vertices’ identifiers and weight, and
- inspected agents: its identifier, vertex, team and internals.
4.2 The Scenario

Score

A step score is calculated on every step, summing up the zones’ scores and the current money. The final score for the team is the sum of these step scores:

\[
\text{score} = \sum_{s=1}^{\text{steps}} (\text{zones}_s + \text{money}_s)
\]

Only probed nodes contribute fully to the zone’s score! (otherwise just 1).

4.3 Programming and running the agents

Starting the MASSim Server

Download and uncompress the MAPC 2011 Package from
http://www.multiagentcontest.org/

You can start the MASSim server by invoking this:

\[
\text{
.}/\text{startServer.sh}
\]

You will then be prompted to choose a simulation.

The monitor

You can start the monitor to observe the simulation:

\[
\text{
.}/\text{startMarsMonitor.sh}
\]

Click on nodes or agent to see more information about the element.

Bear in mind that the monitor shows more information than what agents perceive!
Connections to the server

Agents run independently from the server. They communicate with the server by exchanging XML messages via a socket connection.

The connection is supposed to remain established (or resumed if lost) during the duration of a tournament, which consists of three phases:

1. the initial phase,
2. the simulation phase, and
3. the final phase.

The initial phase

Server

Agent

AUTH-REQUEST

AUTH-RESPONSE

The simulation-phase

Server

Agent

SIM-START

loop: Simulation Step Cycle

REQUEST-ACTION

ACTION

SIM-END

SIM-END
Final phase

Server

Agent

BYE

Simulation state transition

During the simulation step cycle, the state transition is as follows:

1. collect all actions from the agents,
2. let each action fail with a specific probability,
3. execute all remaining attack and parry actions,
4. determine disabled agents,
5. execute all remaining actions,
6. prepare percepts,
7. deliver the percepts.

The environment interface

To facilitate managing the connection to the server, our agents make use of EISMASSim.

- EISMASSim is based on EIS, which is a proposed standard for agent-environment interaction.
- It maps the communication between the MASSim-server and agents, that is sending and receiving XML-messages, to Java-method-calls and call-backs.
- It automatically establishes and maintains connections to a specified MASSim-server.

Running the dummy agents

In the software package we have included a single agent-configuration. It sets up two teams A and B. Each team has 10 agents.

In order to run the dummy agents, navigate to the javaagents/scripts directory and execute

```
$ ./startAgents.sh
```

You will then be asked to select a configuration.
Creating your own agents

In order to create and use your own agents you are required to perform these steps:

1. create a new agent-class that inherits from `massim.javaagents.Agent`,
2. implement a constructor and a couple of required methods,
3. incorporate your new agent-class into the `javaagentsconfig.xml` and, if necessary, adapt the `eismassimconfig.xml`-file,
4. make sure that your new agent-class is in the class-path, and
5. execute.

The abstract agent class already implements some useful data structures and methods for implementing BDI agents, including those for storing goals and beliefs.

The `step`-method is automatically called by the interpreter that executes all agents. It is supposed to return an action, which will then be executed automatically. The `step`-method is the place where you are supposed to add your agent’s intelligence.

Refer to `javaagents.pdf` and `eismassim.pdf`, included with the documentation of the MASSim package, to see the full listings of available methods, percepts and actions.