# Building intelligent agents

## (A4M33MAS/autumn 2010/lecture #3)

**Peter Novák**

**Agent Technology Center, Department of Cybernetics**

**Czech Technical University**

**October 12th 2010**

Czech
Technical
University

agent
technology
center

# Cognitive agents revisited

**cognitive/knowledge intensive agent**

employ cognitive processes, such as knowledge representation and reasoning as the basis for decision making and action selection. I.e., they construct and maintain a *mental state*.

**mental state**

agent's internal explicit representation of the environment, itself, its peers, etc. $\rightsquigarrow$ agent's memory

# The problem

- How to build systems involving mentallistic concepts?
- What are the general principles and guidelines to follow?
- Why building such systems matters?
- What are the main problems we face when building such systems?
- What is the state-of-the-art in this field?

# Lecture outline

1. Motivation & basic concepts

2. Agent-oriented software engineering
   - Introduction
   - Frameworks
   - Tropos methodology

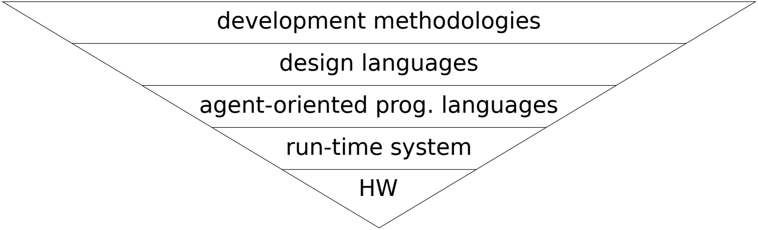
3. Agent-oriented programming
   - Introduction
   - Agent programming languages
   - AgentSpeak(L)/Jason

4. Conclusion

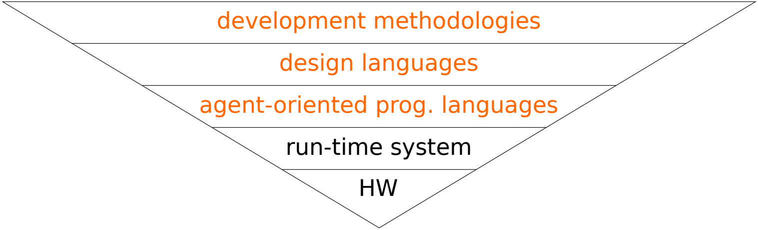
# Motivation & basic concepts

# Agent engineering





# Agent engineering



# Why “agent-oriented”?

- social $\rightsquigarrow$ communication $\rightsquigarrow$ language $\rightsquigarrow$ knowledge representation, reasoning
- autonomy $\rightsquigarrow$ decision making, robust & modular implementation
- proactive $\rightsquigarrow$ opportunistic $\rightsquigarrow$ non-deterministic, parallel
- reactive $\rightsquigarrow$interruptible

**traditional approaches perform poorly in such contexts**

1. interruptions & reactivity $\rightsquigarrow$ exceptions vs. context restore
2. non-determinism vs. structure $\rightsquigarrow$ declarative languages
3. modularity vs. the above $\rightsquigarrow$ elaboration tolerance, compositionality
4. parallelism vs. the above $\rightsquigarrow$ separation vs. interactions
5. KR&R $\rightsquigarrow$ logic-based approaches

# AO software engineering

Highly parallel non-deterministic interruptible behaviours relying on relatively heavy weight knowledge representation and reasoning.

How to model systems in terms of mentalistic concepts?

- knowledge, beliefs
- goals
- obligations
- plans
- roles
- speech-acts

What is the right methodology?

- How to analyse systems?
- How to design systems?

# AO programming languages

Highly parallel non-deterministic interruptible behaviours relying on relatively heavy weight knowledge representation and reasoning.

What is the computational model we should employ for building non-deterministic, parallel and interruptible systems?

- plan encoding
- plan instantiation
- plan execution
- monitoring
- replanning
- failure handling
- reasoning
- integration

What is the system semantics?

- how to: design $\rightsquigarrow$ implementation $\rightsquigarrow$ execution?
- How to verify?

# Agent-oriented software engineering

# What is AOSE?

- methods and tools for supporting development of agent and multi-agent systems oriented software engineering
- modelling languages for the specification of MAS
- techniques for requirements elicitation and analysis
- architectures and methods for designing agents and their organizations
- platforms for implementation and deployment of MAS
- validation and verification methods

# AOSE frameworks

**Modelling frameworks:**

- Tropos
- MaSE
- AUML
- AML
- ...

**Methodologies:**

- Tropos
- Gaia
- Prometheus
- MaSE
- ...

**Special purpose methodologies & modelling tools directed towards:**

- emergent systems
- mobile agents
- swarm intelligence

# Tropos: overview

Tropos is an agent-oriented software engineering (AOSE) methodology that covers the whole software development process.

- requirements driven software development approach ⇝ exploits goal analysis and actor dependencies analysis
- covers also the very early phases of requirements analysis ⇝ deeper understanding of the environment & interactions between software and human agents
- spans from early early analysis down to agent-oriented programming languages issues
- uses mentalistic notions (agent, role, goals, plans, etc.) ⇝ from early analysis down to the actual implementation.

# Tropos language

**Basic concepts:**

- Actor
  - intentional entity: role, position, agent (human or software)
  - agent is an *actor* which occupies a *position* covering (several) *roles* played by the agent
- Goal
  - strategic interest of an actor
  - is associated to an actor.
    - *hard*: clear satisfaction criteria
    - *soft*: qualitative “soft” cirteria
- Task
  - a *course of action* (plan/process) associated with a goal and used to satisfy it by *execution*

# Tropos language (cont.)

**Basic concepts (cont.):**

- Resource
  - physical, or informative *non-intentional entity*
  - can be *used*, *produced*, or *shared*
- Social dependency (between two actors)
  - one actor depends on another to accomplish a goal, execute a task, or deliver a resource
  - the content can be a goal/task/resource

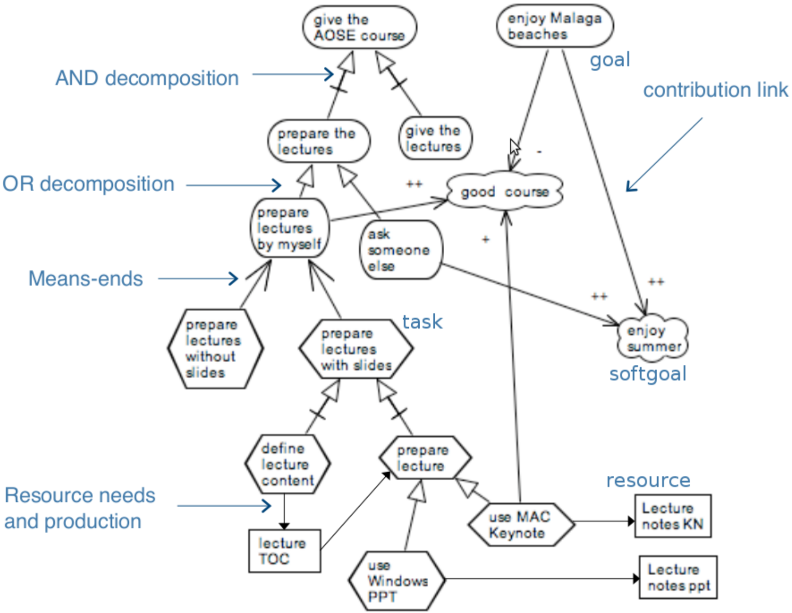
# Tropos language (cont.)



**Basic relations between entities:**

- Decomposition
  - AND decomposition
  - OR decomposition
  - goal $\rightsquigarrow$ subgoals
  - task $\rightsquigarrow$ subtasks
- Means-ends
  - a task (mean) used to achieve a goal (end)
- Contribution
  - a goal/task/softgoal contributes to the satisfaction of a softgoal
- Resource need
  - a task/goal needs a resource
- Resource production
  - a task/goal produces a resource

# Example model



# Tropos methodology

**Phases:**

1. Early requirements (*social domain*)
   - socio- and organizational setting is analyzed and the most relevant actors and their relationships are identified
2. Late requirements (*system in the domain*)
   - the system is introduced as a new actor of the social domain and analyzed in terms of *Tropos* concepts
3. Architectural design (*analysis/decomposition*)
   - the *actor system* is designed
   - *subactors* are introduced and goals/task are assigned
   - *agents* are identified
   - agent *capabilities* are identified
4. Detailed design (*detailed design*)
   - capabilities, protocols, and agent's tasks/plan are specified in detail

# Tropos methodology

**Phases:**

1. Early requirements (*social domain*)
   - socio- and organizational setting is analyzed and the most relevant actors and their relationships are identified
2. Late requirements (*system in the domain*)
   - the system is introduced as a new actor of the social domain and analyzed in terms of *Tropos* concepts
3. Architectural design (*analysis/decomposition*)
   - the *actor system* is designed
   - *subactors* are introduced and goals/task are assigned
   - *agents* are identified
   - agent *capabilities* are identified
4. Detailed design (*detailed design*)
   - capabilities, protocols, and agent's tasks/plan are specified in detail

# Tropos methodology

**Phases:**

1. Early requirements (*social domain*)
   - socio- and organizational setting is analyzed and the most relevant actors and their relationships are identified
2. Late requirements (*system in the domain*)
   - the system is introduced as a new actor of the social domain and analyzed in terms of *Tropos* concepts
3. Architectural design (*analysis/decomposition*)
   - the *actor system* is designed
   - *subactors* are introduced and goals/task are assigned
   - *agents* are identified
   - agent *capabilities* are identified
4. Detailed design (*detailed design*)
   - capabilities, protocols, and agent's tasks/plan are specified in detail

# Tropos methodology

**Phases:**

1. Early requirements (*social domain*)
   - socio- and organizational setting is analyzed and the most relevant actors and their relationships are identified
2. Late requirements (*system in the domain*)
   - the system is introduced as a new actor of the social domain and analyzed in terms of *Tropos* concepts
3. Architectural design (*analysis/decomposition*)
   - the *actor system* is designed
   - *subactors* are introduced and goals/task are assigned
   - *agents* are identified
   - agent *capabilities* are identified
4. Detailed design (*detailed design*)
   - capabilities, protocols, and agent's tasks/plan are specified in detail

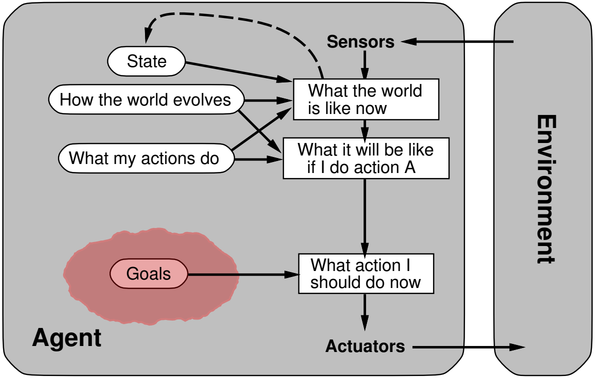
# Tropos methodology

**Phases:**

1. Early requirements (*social domain*)
   - socio- and organizational setting is analyzed and the most relevant actors and their relationships are identified
2. Late requirements (*system in the domain*)
   - the system is introduced as a new actor of the social domain and analyzed in terms of *Tropos* concepts
3. Architectural design (*analysis/decomposition*)
   - the *actor system* is designed
   - *subactors* are introduced and goals/task are assigned
   - *agents* are identified
   - agent *capabilities* are identified
4. Detailed design (*detailed design*)
   - capabilities, protocols, and agent's tasks/plan are specified in detail

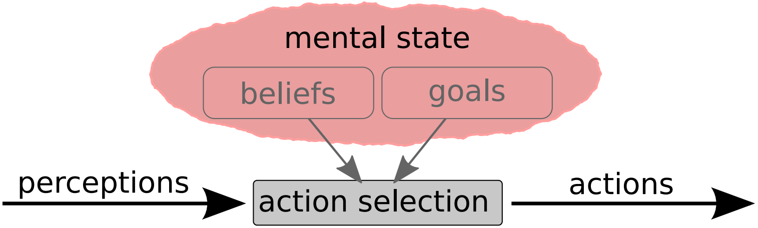
# Agent-oriented programming

# Goal-oriented agents




goals + state + actions' consequences $\rightsquigarrow$ action selection

# Structure of cognitive agents



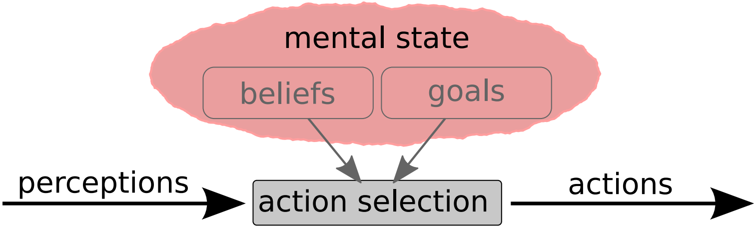

**beliefs** a database of agent's information about itself, the world (environment), other agents, etc.
⇝ NOW

**goals** description of states the agent "wants" to bring about
⇝ FUTURE

**How to select actions leading from NOW to the FUTURE ?**

⇝ **Planning!!!**

# Structure of cognitive agents




**beliefs** a database of agent's information about itself, the world (environment), other agents, etc.
⇝ NOW

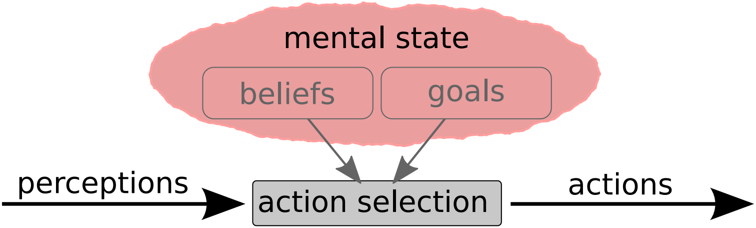
**goals** description of states the agent "wants" to bring about
⇝ FUTURE

**How to select actions leading from NOW to the FUTURE ?**

⇝ **Planning!!!**

# Structure of cognitive agents






**beliefs** a database of agent's information about itself, the world (environment), other agents, etc.
⇝ NOW

**goals** description of states the agent "wants" to bring about
⇝ FUTURE

**How to select actions leading from NOW to the FUTURE ?**

⇝ **Planning!!!**

# Planning

## Definition (planning)

... is the process of generating (possibly partial) representations of future behavior prior to the use of such plans to constrain or control that behavior. The outcome is usually a set of actions, with temporal and other constraints on them, for execution by some agent or agents.

(The MIT Encyclopedia of the Cognitive Sciences)

## plan - execute - monitor cycle

1. plan from the current state to a goal state(s)
2. sequentially execute actions from the plan
3. monitor success of action execution
   - in the case of action failure, (re-)plan again (goto 1)

# The issue with planning

*to arrive to a valid plan, in the worst case, the planner has to explore all the possible action sequences!!!*

$\rightsquigarrow$ high computational complexity ($\approx$PSPACE) ☹

**speed of planning vs. environment dynamics**

planning $\overset{speed}{\succ}$ environment can perform relatively well

planning $\overset{speed}{\prec}$ environment can lead to fatal inefficiencies

$\rightsquigarrow$ the system "suffocates" in (re-)planning

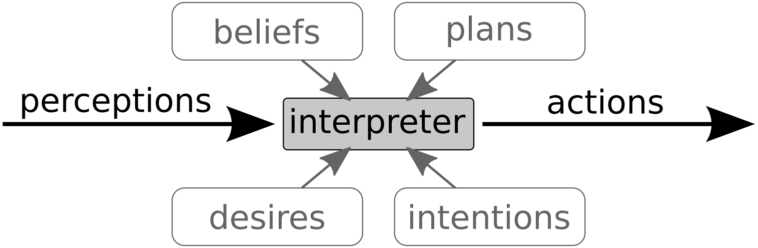
# A way out: BDI

**Structural decomposition:**

- ***(B)eliefs:*** agent's static information about the world
- ***(D)esires:*** situations the agent wants to bring about
- ***(I)ntentions:*** courses of action, plans

**System dynamics:**

- ***reactive planning***: instead of plan-execute-monitor cycle, select partial plans reactively on the ground of the current state of the world, beliefs and goals



# Agent-oriented programming

## Agent-oriented programming

Promotes programming with mentalistic notions and intentional stance as an abstraction. Provides a realization of the BDI agent architecture in pragmatic programming languages.

AOP system:

1. a logical system for *mental states*
2. an interpreted *programming language*
3. an *'agentification'* process

# Agent-oriented programming

## Agent-oriented programming

Promotes programming with mentalistic notions and intentional stance as an abstraction. Provides a realization of the BDI agent architecture in pragmatic programming languages.

AOP system:

1. a logical system for *mental states*
2. **an interpreted *programming language***
3. an *'agentification'* process

# What can APLs do for us?

1. mentalistic abstractions for agent system specification
   - beliefs, desires, intentions, plans, practical reasoning rules, etc.,
   - operationalization of the BDI architecture
   - *tools for encoding the system dynamics*
2. agent-oriented language semantics
   - syntax & model of execution
   - loosely corresponds to temporal modal logics
3. means to tackle the pro-activity vs. reactivity problem
   - deliberation/planning vs. handling events & interruptions $\rightsquigarrow$ hybrid architectures

# Historical overview

*Hybrid architectures:* *– incomplete –*

**1987: PRS** (Georgeff and Lansky)
1988: IRMA (Bratman, Israel and Pollack)
**1991: Abstract BDI architecture** (Rao and Georgeff)
1994: INTERRAP (Müller and Pischel)

---

*Agent-Oriented Programming Languages:* *– incomplete –*

**1990: AGENT-0** (Shoham)
**1996: AgentSpeak(L)** (Rao)
1996: Golog (Reiter, Levesque, Lesperance)
1997: 3APL (Hindriks et al.)
1998: ConGolog (Giacomo, Levesque, Lesperance)
2000: JACK (Busetta et al.)
2000: GOAL (Hindriks et al.)
2002: Jason (Bordini, Hubner)
2003: Jadex (Braubach, Pokahr et al.)
2008: BSM/Jazzyk (Novák)
2008: 2APL (Dastani)

# The landscape



## BDI programming systems

*Theoretically oriented*

- (−) declarative languages built from scratch ⇝ new syntax
- (+) clear theoretical properties ⇝ verification
- (+) declarative KR techniques
- (−) no integration with external/legacy systems

AgentSpeak(L), 3APL, 2APL, GOAL, CAN, etc.

*Engineering approaches*

- (+) layer of specialised language constructs over a robust mainstream programming language (Java) ⇝ code re-usability
- (−) host language semantics
- (−) KR in an imperative language
- (+) easy integration with external systems and environments

JACK, Jadex

# AgentSpeak(L)/Jason

- programming language for BDI agents
- notation based on logic programming
- AgentSpeak(L) $\rightsquigarrow$ an abstract programming language
- Jason $\rightsquigarrow$ operational semantics for AgentSpeak
- incorporates Prolog for reasoning about beliefs
- various pragmatic extensions like external actions (Java)
- also a platform for developing multi-agent systems

- provides a clean & simple implementation of agent concepts

http://jason.sourceforge.net/

# AgentSpeak syntax

## Beliefs

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

`location(Object, Coordinate), at(Coordinate)`

## Goals

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

achievement goals: achieve a state

`!write(book), !at(Coordinate)`

test goals: retrieve information from the belief base

`?at(location(Object, Coordinate))`

# AgentSpeak syntax (cont.)

Agent reacts to events by executing plans.

agent program = set of rules

`triggering_event : context` $\leftarrow$ `body.`

***triggering event:*** (perceived) change/event to handle

`+b, -b, +!g, -!g, +?g, -?g` $\leadsto$ implicit goals!

***context:*** circumstances in which the plan can be used

logical formula ($\wedge, \vee, \neg$)

***body:*** 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

$\leadsto$ a means to an end

# AgentSpeak mechanisms

**Triggering events:**

- +b (add belief)
- -b (delete belief)
- +!g (add a-goal)
- -!g (delete a-goal)
- +?g (add test-goal)
- -?g (delete test-goal)

**Intention = stack of:**

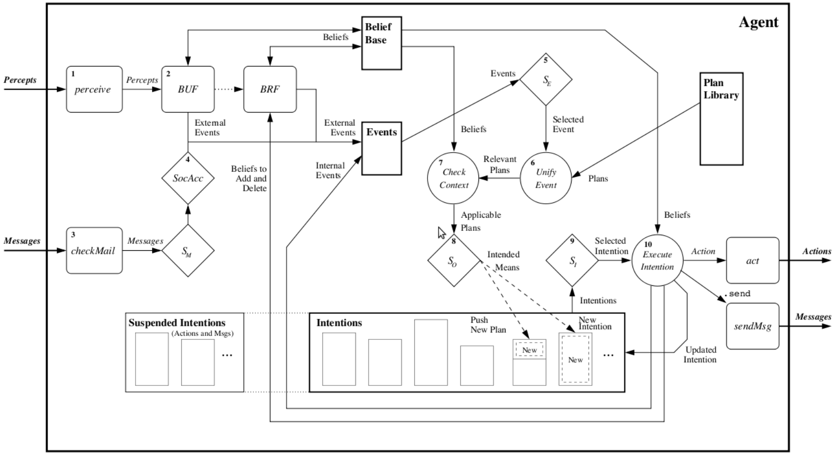
- environment actions
- achievement goals
- test goals
- internal actions
- expressions
- mental notes

# AgentSpeak reasoning cycle



1. perceive the environment
   - receive communication from other agents
   - select 'socially acceptable' messages
2. update the belief base
3. select an event to handle
4. retrieve all relevant plans
5. determine the applicable plans
6. select one applicable plan
7. select an intention for further execution
8. execute one step of an intention

# AgentSpeak interpreter



# Jason (example)

```
+green_patch(Rock)
    : not battery_charge(low)
    ← ?location(Rock,Coordinates);
      !at(Coordinates);
      !examine(Rock).

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

# Contingency plans & failure



Contingency plans $\leadsto$ multiple rules + single triggering event:

```
+!at(Coords)
  : not at(Coords) & safe_path(Coords)
  ← move_towards(Coords); !at(Coords).
+!at(Coords)
  : not at(Coords) & no_safe_path(Coords) & not storm
  ← fly_towards(Coords); !at(Coords).
+!at(Coords)
  : not at(Coords) & very_bad_weather
  ← ask_for_teleport(Coords); ....
```

A plan failure triggers a goal-deletion event:

```
-!at(Coords)
  : very_bad_weather
  ← !wait_for_good_weather.
```

# Contingency plans & failure

Contingency plans ⇝ multiple rules + single triggering event:

```
+!at(Coords)
  : not at(Coords) & safe_path(Coords)
  ← move_towards(Coords); !at(Coords).
+!at(Coords)
  : not at(Coords) & no_safe_path(Coords) & not storm
  ← fly_towards(Coords); !at(Coords).
+!at(Coords)
  : not at(Coords) & very_bad_weather
  ← ask_for_teleport(Coords); ....
```

A plan failure triggers a goal-deletion event:

```
-!at(Coords)
  : very_bad_weather
  ← !wait_for_good_weather.
```

# Internal actions

Internal actions serve:

1. as a glue between Jason and external legacy code (Java)
   `map.get_coords(Rock, Coords),`
   `.send(...), .print(...)`
2. as a means to manually steer the deliberation cycle
   `.desires(...), .drop_desires(...)`

```
+green_patch(Rock)
  : ~battery_charge(low) & .desires(at(_))
  ← .drop_desires(at(_));
    map.get_coords(Rock, Coords);
    !at(Coords);
    !examine(Rock).
```

# MAS specification

**Putting it all together:**

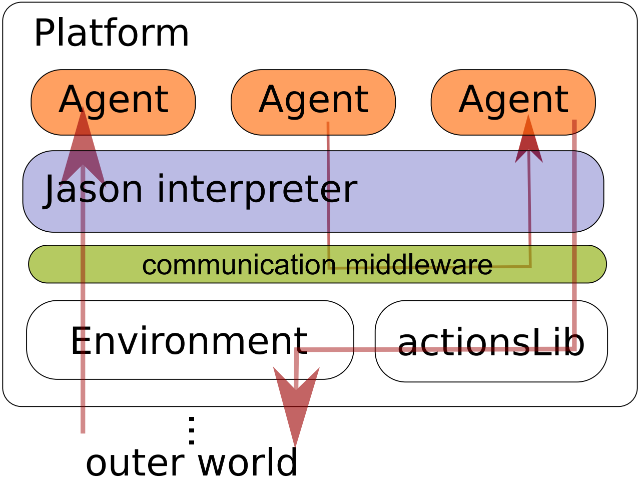
MAS = agents + communication + environment

```
MAS mars_exploration_system {
    /* communication infrastructure (built-in) */
    infrastructure: Centralised

    /* interface to the environment (Java class) */
    environment: MarsEnv

    /* agents in the MAS (Jason agents) */
    agents: Spirit; Opportunity; Beagle2;
}
```

# Jason system architecture



# Conclusion

# Summary

1. Motivation & basic concepts
2. Agent-oriented software engineering
   - Introduction
   - Frameworks
   - Tropos methodology
3. Agent-oriented programming
   - Introduction
   - Agent programming languages
   - AgentSpeak(L)/Jason
4. Conclusion

# Final thoughts

## Agent-oriented software engineering

... provides a useful view on complex distributed systems. In the core, there is the idea of loose coupling of components and a strong emphasis on autonomy.

## Agent-oriented programming

... is just one of the ways to tackle the problem of reactivity vs. deliberation. There are other as well: hybrid robotic architectures, situation/fluent-calculus based approaches, cognitive architectures, etc.

- BDI architecture $\rightsquigarrow$ modelling smart robotic and multi-robot systems

...both fields are a subject to an active on-going research, so the story is far from over.

# Final thoughts

## Agent-oriented software engineering

... provides a useful view on complex distributed systems. In the core, there is the idea of loose coupling of components and a strong emphasis on autonomy.

## Agent-oriented programming

... is just one of the ways to tackle the problem of reactivity vs. deliberation. There are other as well: hybrid robotic architectures, situation/fluent-calculus based approaches, cognitive architectures, etc.

- BDI architecture $\rightsquigarrow$ modelling smart robotic and multi-robot systems

...both fields are a subject to an active on-going research, so the story is far from over.

# The end

## Thank you for your attention.

## Questions?

Resources:

- ČVUT CourseWare: A4M33MAS
- http://www.troposproject.org/
- http://jason.sourceforge.net/