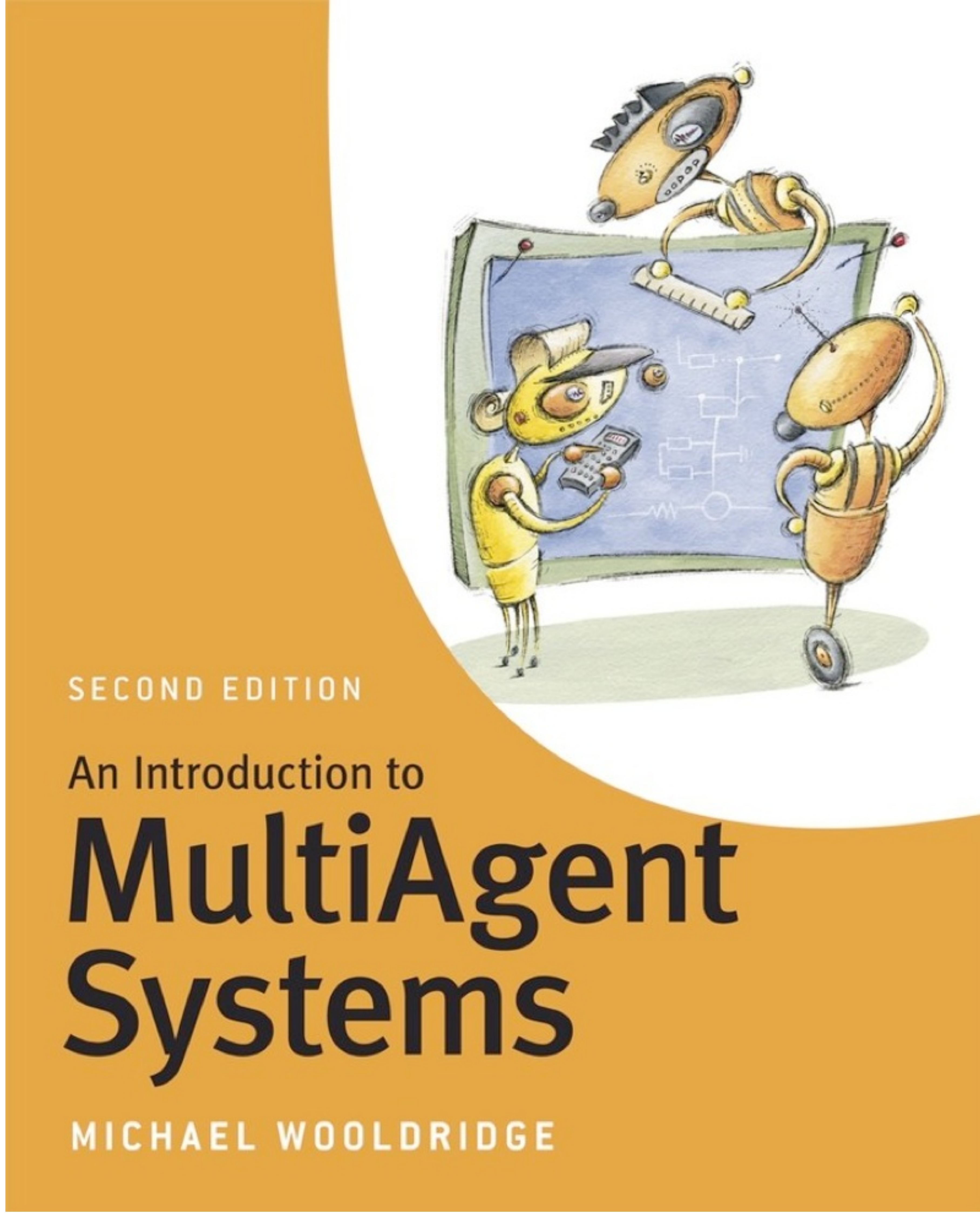
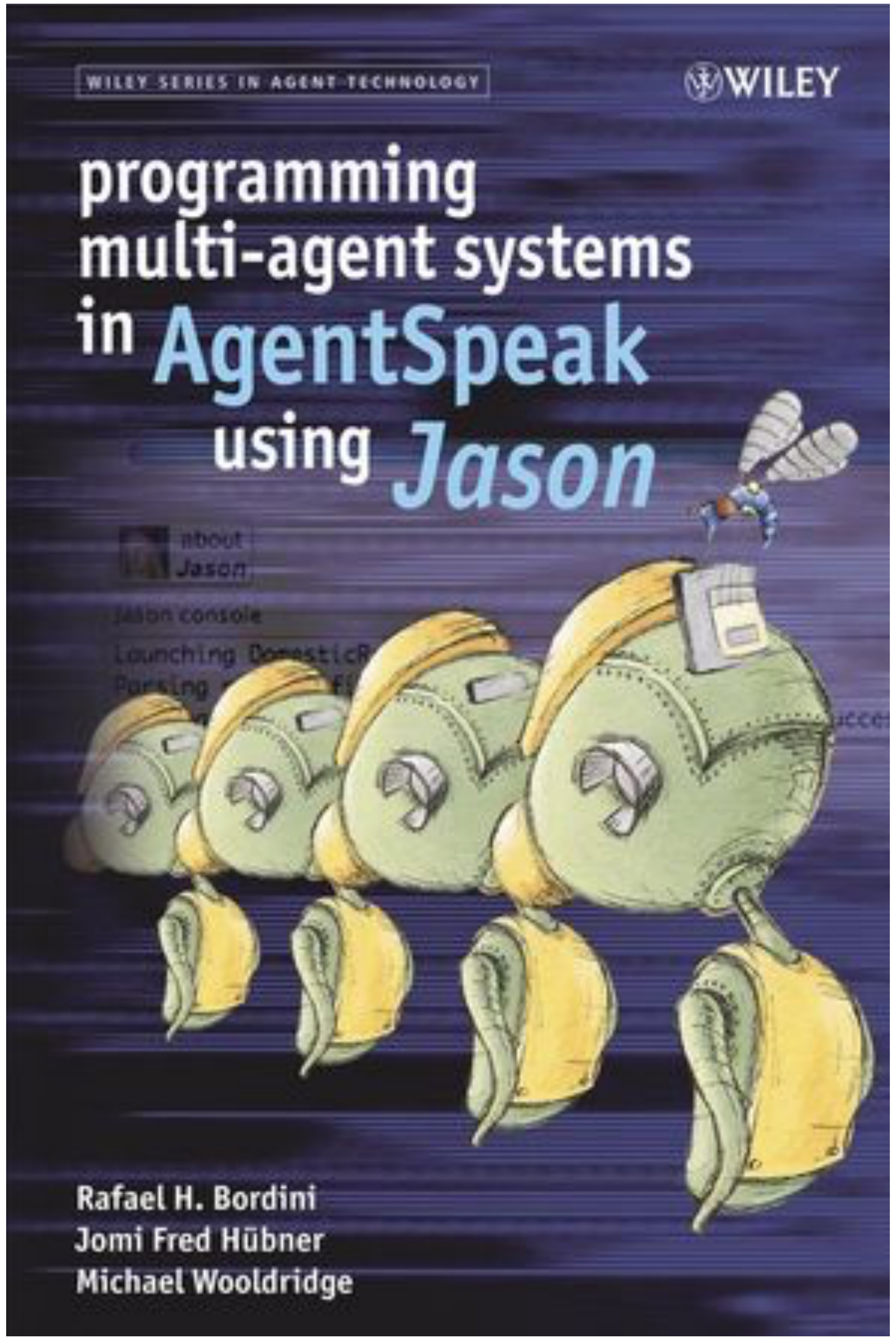
COMP310
Multi-Agent Systems
Chapter 4a - Jason and AgentSpeak

Dr Terry R. Payne
Department of Computer Science
UNIVERSITY OF LIVERPOOL



The Jason Agent Programming Language

- In this lecture we will look at a BDI-based Agent Programming Language
  - AgentSpeak (originally developed by Rao, 1996)
- Jason is an open-source interpreter for an extended version of AgentSpeak
  - Based on:
    - PRS architecture
    - BDI logics
    - Logic Programming (Prolog)
  - Became the language of choice for Multi-Agent Programming Contest

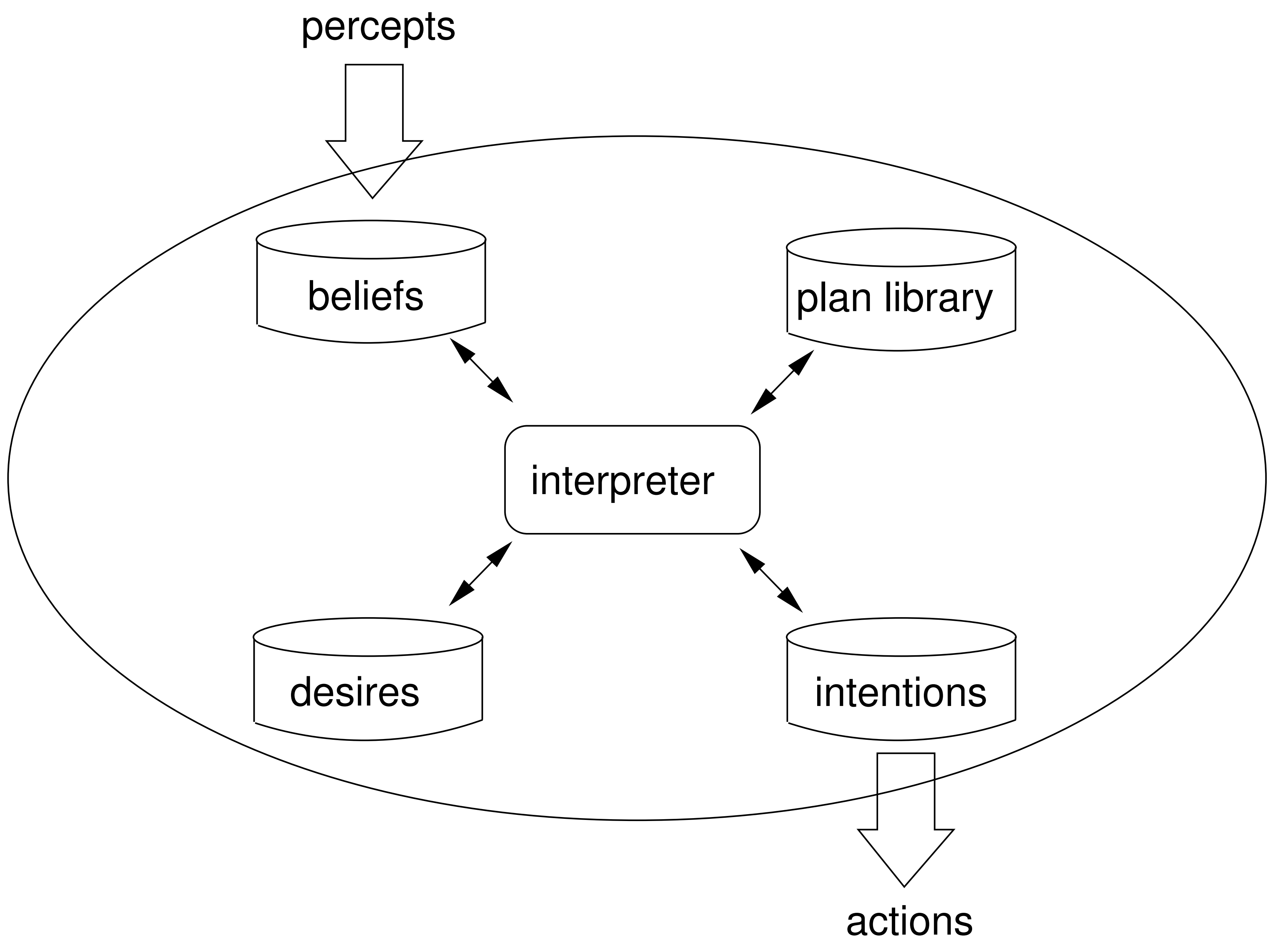


Programming Languages for Agents

- Desirable properties for Agent Programming Languages
  - Support delegation at the level of goals
    - Focusing on the ***what***, not ***how***
  - Support goal-directed problem solving
    - Agents acting to ***achieve*** their delegated goals
  - Should be responsive their environment
    - Environment should be ***compatible with other frameworks***, or simulators
  - Should support knowledge-level communication and cooperation
    - Exchange ***beliefs***, ***goals*** and ***plans***

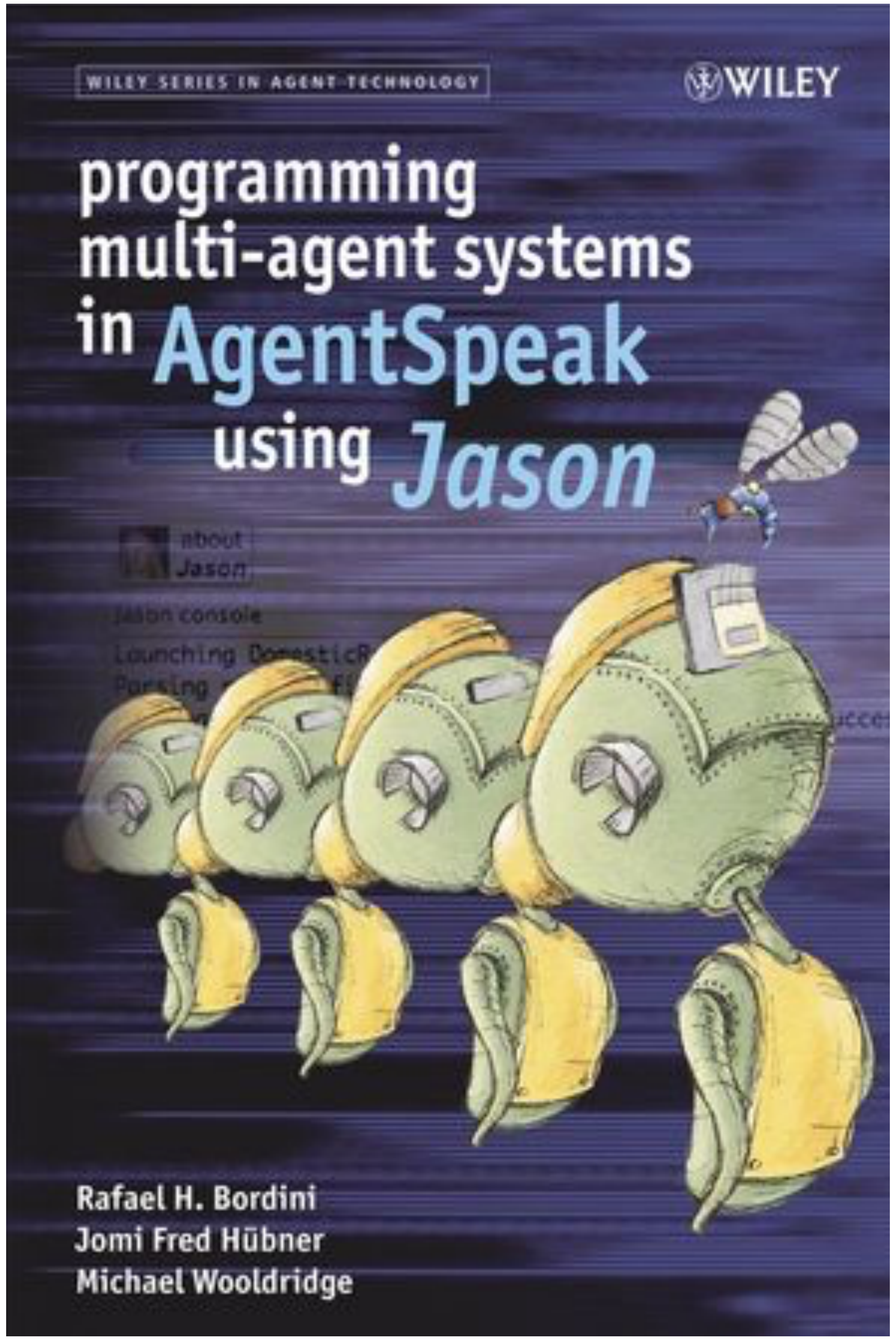
AgentSpeak as an Agent Architecture

- The variant of AgentSpeak interpreted by Jason is based on a BDI architecture (similar to PRS)
- A Reactive Planning System
  - Permanently running, responding to events by executing plans
  - Actions then affect the environment
  - The agent reasons about how to act to achieve its goals
- Practical Reasoning
  - Achieved through the use of a Plan Library
  - Similar to that used by PRS



Jason

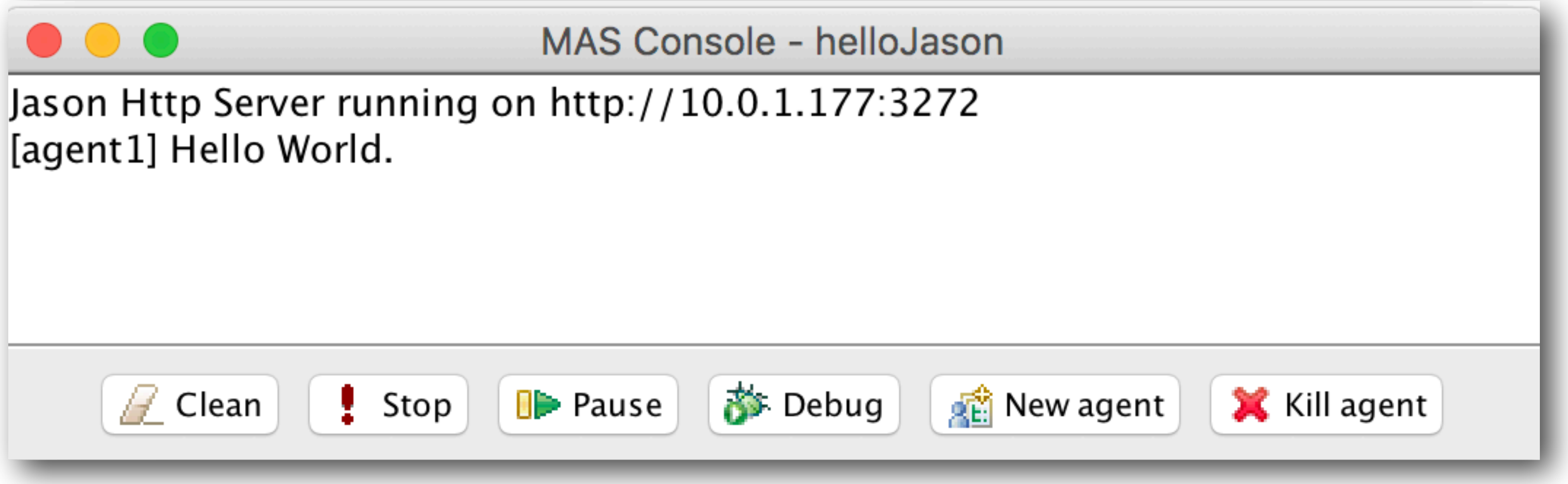
- Developed by Jomi F. Hübner & Rafael H. Bordini
  - Source is available from:
    - https://sourceforge.net/projects/jason/
- Implements the operational semantics of an extended version of AgentSpeak
  - Highly customisable, with extensions to other Agent Frameworks (including JADE)
  - Optional programmable Environment (Java)
- Book published by John Wiley & Sons.
  - http://jason.sf.net/jBook/



Hello World (in AgentSpeak)

- The iconic “Hello World”
  - Line 4 - we create the ***belief*** `started`
    - Here we have a symbol, but beliefs can also be predicates
  - Line 7 - we have a plan that is triggered by the addition of the belief `started`
    - Means “…*when you come to believe ‘started’, then print some text…*”
    - The “+” here signifies when you ***acquire*** the belief…
- Plans can have contexts
  - Different plans may be triggered for a belief, depending on the context

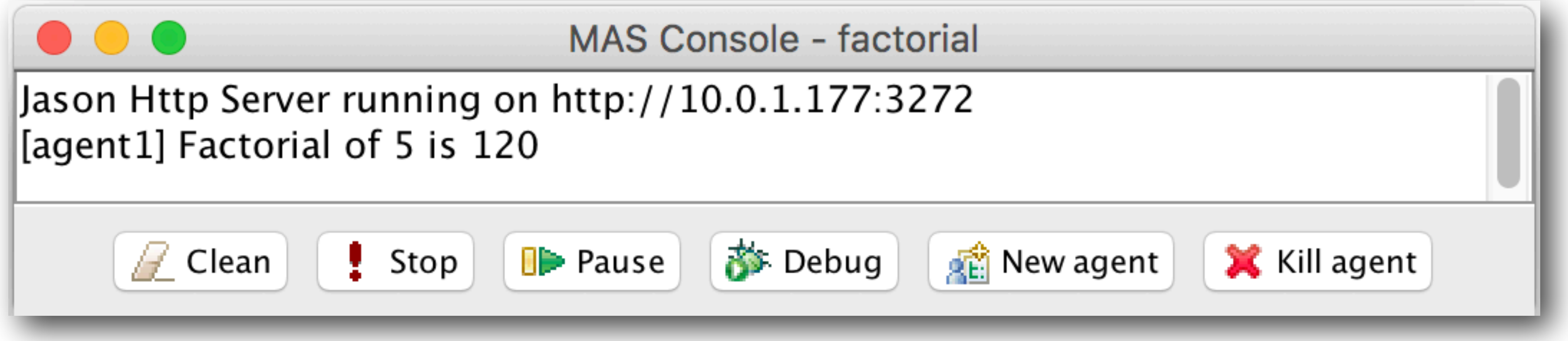
```
1. // Taken from Programming Multi-Agent
2. // Systems in AgentSpeak using Jason
3. /* my initial belief */
4. started.
5.
6. /* Plans */
7. +started <- .print("Hello World.”).
```



Factoral (in AgentSpeak)

- This example uses ***goals***
  - Line 2 - add the goal `print_fact(5)`
    - Here we have a symbol, but beliefs can also be predicates
  - Line 5 - the plan for `print_fact(5)`
    - The upper-case `N` is a variable, instantiated by the goal (i.e. `N=5`)
    - Line 6 - create the goal `fact(N,F)` which will instantiate the variable `F`
    - Line 7 - when achieved, print the values of `N` and `F`
  - Line 9 - the plan for `fact(N,1)`
    - Only triggers for the context `N == 0`
    - Instantiates the value of the second variable (i.e. `F`) to be 1
      - Means that the value is 1 for the factorial of 0
  - Line 11 - the plan for `fact(N,F)`
    - Only triggers for the context `N > 0`
    - Generates the factorial of `N-1` (by creating a new goal)
    - Then instantiates the new value of `F`

```
1.  /* Initial achievement goal */
2.  !print_fact(5).
3.
4.  /* Plans */
5.  +!print_fact(N)
6.      <- !fact(N,F);
7.          .print ("Factorial of ", N, " is ", F).
8.
9.  +!fact(N,1) : N == 0.
10.
11.+!fact(N,F) : N > 0
12.    <- !fact(N-1, F1);
13.         F = F1 * N.
```



AgentSpeak as an Agent Architecture

- There are three main language constructs in AgentSpeak:
  - Beliefs
  - Goals
  - Plans

- The architecture of AgentSpeak has four main components:
  - Belief Base
  - Plan Library
  - Set of Events
  - Set of Intentions

Beliefs

- Beliefs are simple Prolog statements, stored in a ***Belief Base***.

- Two kinds of statement.
  - Facts
    - Simple propositions (prolog atoms) or predicates relating propositions
  - Axioms (or rules)
    - Allow inference of new beliefs from existing ones
    - Instantiates the values of logical variables through ***unification***

- Modalities of Truth
  - Beliefs refer to what they agent believes about the world, not ground truth

Facts

Propositions or Predicates

```
starting
academic(terry)
teaches_comp310(terry)
parent(terry, alessandro)
```

Beliefs can be negated

```
~starting
~teaches_comp101(terry)
```

The symbol ~ should read not

Note that atoms and predicates start with a lower case letter

Variables, Rules and Unification

- Belief Axioms look a lot like ***rules*** in Prolog.
  - `child(X, Y) :- parent(Y, X).`
  - Read the rule
    - a :- b as "a holds if b holds" or "if b then a".
- These axioms allow agents to ***infer*** new predicates
  - For example: `parent(bob, jane)` matches `parent(Y, X)` if `Y` = `bob`, and if `X` = `jane`
  - The agent can then infer `child(jane, bob)`

- Rules are allowed to be more complex than this.
  - For example: `grandparent(X, Z) :- parent(X, Y) & parent (Y, Z).`
  - The "&" represents ***conjunction***, and is what we usually mean by "and".
  - So, given:
    - `parent(eric, bob) parent(bob, jane).`
  - the agent can infer:
    - `grandparent(eric, jane)`

What can be inferred?

- Rules and Axioms
  - grandparent(X, Z) :- parent(X, Y) & parent (Y, Z).
  - child(X, Y) :- parent(Y, X).
  - son(X, Y) :- child(X, Y) & male(X).
  - daughter(X, Y) :- child(X, Y) & female(X).
  - parent(eric, hilary)
  - parent(hilary, jane)
  - parent(hilary, david)
  - female(jane)
  - male(david)
- Possible Inference?
- Note that we don't know the gender of hilary

What can be inferred?

- Rules and Axioms
  - grandparent(X, Z) :- parent(X, Y) & parent (Y, Z).
  - child(X, Y) :- parent(Y, X).
  - son(X, Y) :- child(X, Y) & male(X).
  - daughter(X, Y) :- child(X, Y) & female(X).
  - parent(eric, hilary)
  - parent(hilary, jane)
  - parent(hilary, david)
  - female(jane)
  - male(david)

- Possible Inference
  - grandparent(eric, jane)
  - child(hilary, eric)
  - child(jane, hilary)
  - child(david, hilary)
  - son(david, hilary)
  - daughter(jane, hilary)

- Note that we don't know the gender of hilary

Belief Annotations

- Logical Formulae in Jason can be annotated
  - Strongly associates additional information to a belief
    - Represented as Prolog lists
  - More elegant than stating additional beliefs, or having beliefs of beliefs
    - Facilitates organisation and management of beliefs
      - Most annotations mean nothing to the interpreter
      - However, java can be used to manage the ***belief base***

Belief Annotations

- Annotated predicate

  ps(t_1,...,t_n)[a_1,...,a_m]

  - Where a_i are first order terms

- All predicates in the belief base have a special annotation:

  source(s_i)

  - where $s_i \in$ {self,percept} ∪ agentId

Examples

busy(john) [expires(autumn)]

The agent believes that john is busy, but when autumn starts, this belief no longer holds

~colour(box1,white) [source(percept)]

The agent believes, based on perceiving the world, that the colour of box1 is not white

liar(bob) [source(self),degOfCert(0.2)]

The agent believes bob is a liar, based on its own evidence, but with only a 0.2 degree of certainty

Belief Annotations

- Source annotations have a specific meaning within Jason
  - Perceptual information `[source(percept)]`
    - If an agent acquires beliefs from sensing its environment, then it is annotated as a percept
  - Communication [source(*agentID*)]
    - If agents communicate, then beliefs that are shared are annotated with the sender's ID
  - Mental Notes `[source(self)]`
    - Beliefs that are added by the agent itself can help it remember past activities. These are things that the agent can use to remind itself in the future.

- Beliefs given to the agents without annotations are assumed to be mental notes
  - And are annotated as such!

Belief Dynamics

- By perception
  - Beliefs annotated with `source(percept)` are automatically updated according given any perceptions of the agent
- By intention
  - The plan operators **+** and **-** can be used to add and remove beliefs annotated with `source(self)`
    - mental notes
- By communication
  - When an agent receives a `tell` message, the content is a new belief annotated with the sender of the message

Belief Dynamics

By intention:

```
+friend(bob);
    // adds friend(bob)[source(self)]
-friend(eric);
    // removes friend(bob)[source(self)]
```

By communication:

```
.send(alice, tell, friend(bob));  // sent by ian
    // adds friend(bob)[source(ian)]
    // to ian's set of beliefs
etc
```

Not & Strong Negation

- The problem with the closed world assumption
  - It assumes that ***anything that is not believed to be true must be false***
  - But what if you want to refer to:
    - Things the agent believes to be ***true***
    - Things the agent believes to be ***false***
    - Things the agent ***doesn't have beliefs*** about (whether or not they are true or false)?
- Logically, ***not*** only allows the negation of a formula
  - We can check if something is ***true***, or if something is ***not true (i.e. false)***
  - But this says ***nothing about what it is that is believed***!

Strong Negation

- The operator '**~**' represents strong negation
  - i.e. an agent explicitly believes something to be false
- The operator '***not***' represents weak negation
  - i.e. logically, an expression is not true, and therefore an agent doesn't have the belief
    - Whether or not the belief is that something is true or false

Beliefs

Consider the following beliefs:

```
colour(box1, blue)[source(bob)]
~colour(box1, white)[source(john)]
```

The agent believes that the colour of box1 is blue, and that the colour of box one is not white.

Now consider these negated beliefs:

```
not shape(box1, cube)[source(percept)]
not ~shape(box1,sphere)[source(self)]
```

The agent does not have the belief that the shape of box1 is a cube. But conversely, it does not have a belief that the shape isn't a sphere, either.

More on rules

- Consider the rules opposite:
  - The first states that the most likely colour of some object `B` is the colour the agent deduced earlier, or the one it perceived
  - If this fails, then the likely colour of `B` should be:
    - the one with the highest degree of certainty associated with it
    - Provided that there is strong evidence (i.e. that the agent believes) that object `B` is not colour `C`.
- This is an example of ***theoretical reasoning***
  - In Jason, ***practical reasoning*** is achieved through plans

Rules for 'likely_colour'

```
likely_colour(C,B)
   :- colour(C,B)[source(S)] &
      (S == self | S == percept).

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

Goals

- Goals represent ***the properties of the state of the world*** that the agent ***wishes to bring about***

- Two types of goals:
  - ***Achievement goals*** (i.e. to do): !g
    - This is a goal the agent wants to bring about
    - The goal is not currently believed to be true, and therefore the agent will aim to resolve this
      - Typically involves executing an associated plan
  - ***Test goals*** (i.e. to know): ?g
    - More similar to Prolog goals (or queries) - the agent wants to check if the goal is true

Achievement Goals

`!own(house)`

The agent will try to bring about the state where the belief `own(house)` is true.

Test Goals

`?teaches(terry, Module)`

The agent needs to establish a value for the variable Module that makes this belief true.

Often this is used to unify a variable, but in certain circumstances, test goals may also lead to the execution of plans.

Plans

- ***Plans*** are recipes for action, representing the agents know-how
  - ***Intentions*** are plans instantiated to achieve some goal
- Each plan has three distinctive parts:
  - The ***triggering event*** denotes the events the plan is meant to handle
  - The ***context*** represents the circumstances in which the plan can be used
  - The ***body*** is the actual plan to handle the event if the context is believed true at the time a plan is being chosen
  - Plans can also have an ***optional label***
- When the trigger happens, test the context, and if it is true, then execute the plan

Plan Syntax

triggering event : context <- body.

Plan Syntax (with label)

@label te : context <- body.

Triggering Events

- Events happen as a consequence of changes in the agents beliefs or goals
  - An agent reacts to events by executing plans
  - Types of plan triggering events:

`+b`	Belief addition
`-b`	Belief deletion
`+!g`	Achievement-gaol addition
`-!g`	Achievement-gaol deletion
`+?g`	Test-goal addition
`-?g`	Test-goal deletion

Example plans

- A plan that responds to a change in belief.
  - Triggering Event
    - When the belief `green_patch(Rock)` is added.
      - i.e. when you realise that the rock has a green patch
  - Context
    - If battery charge is not low
      - i.e we don't have the belief `battery_charge(low)`
  - Body
    - Find the location of the rock - using a test goal
    - Go to that location - achieve the goal `at(Coordinates)`
    - Examine the rock - achieve the goal `examine(Rocks)`

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

AgentSpeak Plans

- Plans are a bit like STRIPS actions:
  - Preconditions (i.e. the context)
  - What you do (i.e. the plan body)
- However, plans also contain more than one action
- Plans are also a bit like STRIPS plans
  - Sequence of things to do…
  - …but they also have preconditions and subgoals.

Example plans

- A plan that responds to the addition of a goal.
  - Triggering Event
    - Get to a set of coordinates - i.e. achieve the goal `!at(Coordinates)`
  - Context
    - If not at the coordinates (i.e. we don't have that belief)…
    - …and there is the belief that there is not an unsafe path to the coordinates
  - Body
    - Move towards the coordinates
    - This would result in the ***action*** `move_towards(Coordinates)` being called in the ***Environment***
    - Assert (again) the goal of being at the coordinates

```
+!at(Coordinates) : not at(Coordinates)
            & ~ unsafe_path(Coordinates)
   <- move_towards(Coordinates);
      !at(Coordinates).
```

- This recursive setting allows for plans that partially achieve the goal.

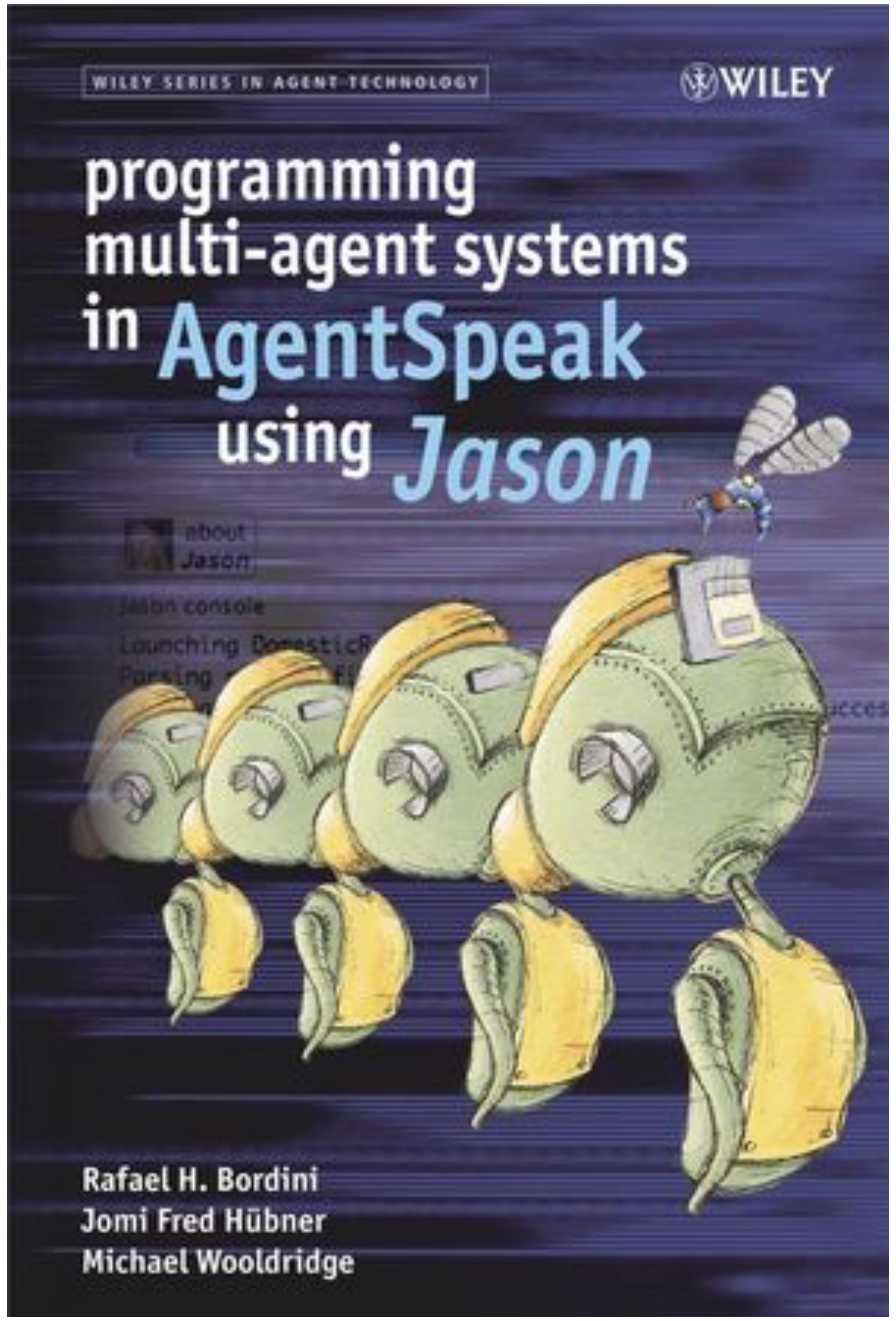
Actions

- An Agent needs to be able to act within an environment
  - Note that actions in an AgentSpeak program are logical statements (predicates)
    - A predicate in the context is interpreted as a ***belief***
    - A predicate in the plan is interpreted as an ***action***
  - Actions are ground predicates
    - i.e. any variables should be instantiated before the action is performed
    - In the plan opposite, the action `move_towards(Coordinates)` results in some method being called in the environment.java object
  - Actions prefixed with the '.' refer to internal actions
    - E.g. '.print' and '.send'

```
+!at(Coordinates) : not at(Coordinates)
            & ~ unsafe_path(Coordinates)
   <- move_towards(Coordinates);
      !at(Coordinates).
```

Internal Actions

- Jason can be used to support advanced BDI agents
  - Including the definition of maintenance and achievement goals
  - The types of commitment (blind, single-minded etc)
  - Several internal actions have been provided to support this
    - ‘.desire’, ‘.intend’, ‘.succeed_goal’, ‘.fail_goal’ etc



- Chapter 8 in Rafaels book provides a number of patterns
  - for defining such goals, commitments etc

Summary

- This lecture introduced the syntax of AgentSpeak
- We discussed its main constructs:
  - beliefs
  - goals
  - plans
- These slides are based on Chapter 3 of Rafael's book on Jason
  - Optional activities will be posted on the website
  - More advanced patterns for Commitment Strategies, an explicitly modelling desired and intentions are also discussed in Chapter 8
  - We'll come back to AgentSpeak later in the module

Class Reading (Chapter 4a):

Anand S. Rao, 1996. AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language. Proceedings of Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW-96)

This paper gives an initial description of the original AgentSpeak(L) languages, as well as its formal properties.