Jason

Programming Multi-Agent Systems using Jason

Jan Toennemann, TU Clausthal
27. June 2017
Table of Contents

Introduction

Jason – an Extension of AgentSpeak

Development of Multi-Agent Systems using Jason

Conclusion

Sources
Introduction

Introduction to Agent-Oriented Programming
A very short Introduction to Prolog
The BDI-Model

Jason – an Extension of AgentSpeak

Development of Multi-Agent Systems using Jason

Conclusion

Sources
Introduction to Agent-Oriented Programming

- Functional program: Input → computing → Output; \( f : I \rightarrow O \)
- Reactive system: state-based, concurrent composition of functional programs
- Agents: reactive system with a degree of autonomy
“A logic program is a set of axioms, or rules, defining relations between objects. A computation of a logic program is a deduction of consequences of the program. A program defines a set of consequences, which is its meaning. The art of logic programming is constructing concise and elegant programs that have the desired meaning.” (The Art of Prolog, p. 47)
Facts and Queries

- Facts:
  - drinks(beer).
  - drinks(water).

- Queries:
  - ? - drinks(beer).
    \% → yes
  - ? - drinks(milk).
    \% → no
  - ? - drinks(X).
    \% → beer
Facts and Queries

- **Facts:**
  
  \[ \text{drinks(beer)}. \]
  \[ \text{drinks(water)}. \]

- **Queries:**

  \[ ? \text{ - drinks(beer). } \% \rightarrow \text{yes} \]
  \[ ? \text{ - drinks(milk). } \% \rightarrow \text{no} \]
  \[ ? \text{ - drinks(X). } \% \rightarrow \text{beer} \]
Relationships and Rules

- Facts and Rules:
  
  drinks(john, water).
  drinks(jeremy, milk).
  drinks(camilla, beer).
  drinks(jeremy, X) :- drinks(john, X).

- Queries:
  
  ? - drinks(camilla, X)
  → X = beer
  ? - drinks(X, gin).
  → no
  ? - drinks(jeremy, X).
  see proof/query tree
Relationships and Rules

- Facts and Rules:
  
  \[
  \text{drinks}(john, \text{ water}). \\
  \text{drinks}(jeremy, \text{ milk}). \\
  \text{drinks}(camilla, \text{ beer}). \\
  \text{drinks}(jeremy, X) :- \text{drinks}(john, X). \\
  \]

- Queries:
  
  ? - drinks(camilla, X) \% \rightarrow X = \text{beer} \\
  ? - drinks(X, gin). \% \rightarrow \text{no} \\
  ? - drinks(jeremy, X). \% \text{see proof/query tree}
Relationships and Rules

The straight lines show how one goal sets up another goal. The curved lines show the alternative values for X for different solutions. The general case is of a sequence of queries that must be satisfied. The subgoals in a query are separated by commas. Prolog begins from left to right attempting to satisfy each query. If a subgoal succeeds, Prolog tries the next one on the right. If a subgoal fails, Prolog goes back to the goal on the left to see if there are any more solutions. So, if we wish to test whether some X is human and honest, the query ?- human(X), honest(X). is executed. This picture might help:

\[ \text{drinks(jeremy, X )} \]
\[ \text{drinks(jeremy, milk) } \]
\[ \text{drinks(john, X )} \]
\[ \text{drinks(john, water) } \]
Agent-Oriented Programming

- Beliefs
- Desires
- Intentions
Reasoning based on the BDI-Model

- reasoning is done in a **control loop**
- simplified graphic of procedural reasoning:

![Diagram](image.png)
Introduction

Jason – an Extension of AgentSpeak
Core Concepts: Beliefs, Goals and Plans
Advanced Features
Basic Example

Development of Multi-Agent Systems using Jason

Conclusion

Sources
General

- Prolog-like syntax, extended by dynamic elements
- + to add to belief base, - to remove
- ! for achievement goals, ? for test goals
- plans consist of a trigger condition, a context and a body
Beliefs and Plans

Recursive computation of the factorial of 5 using Jason:

\[
\text{fact}(0, 1).
\]

\[
+\text{fact}(X, Y) : X < 5
\]
\[
\quad \leftarrow +\text{fact}(X + 1, (X + 1) \ast Y).
\]

\[
+\text{fact}(X, Y) : X == 5
\]
\[
\quad \leftarrow .\text{print}("\text{fact} 5 == ", Y).
\]
Goals

Recursive computation of the factorial of 5 using Jason’s goals:

!print_fact(5).

+!print_fact(N)
  <- !fact(N, F);
      .print("Factorial of ", N, " is ", F).

+!fact(N, 1) : N == 0.

+!fact(N, F) : N > 0
  <- !fact(N - 1, F1);
      F = F1 * N.
Actions and Internal Actions

- Internal actions can be developed in Java and called from agent code
- IAs from standard library are preceded by a .
Actions and Internal Actions

- Internal actions can be developed in Java and called from agent code
- IAs from standard library are preceded by a .

```plaintext
+!leave(home) : not raining & not ~raining
<- !location(window);
  ?curtain_type(Curtains);
  open(Curtains).

+!leave(home) : not raining & not ~raining
<- .send(mum, askIf, raining).
```
Inter-Agent Communication

- agents can communicate using
  `.send(agent, action, content)` and
  `.broadcast(action, content)`

- possible actions are `tell`, `untell`, `achieve`, `unachieve`, `askOne`, `askAll`, `tellHow`, `untellHow` and `askHow`
Inter-Agent Communication

- agents can communicate using `.send(agent, action, content)` and `.broadcast(action, content)`
- possible actions are `tell`, `untell`, `achieve`, `unachieve`, `askOne`, `askAll`, `tellHow`, `untellHow` and `askHow`
- communication triggers events at the receiving agent when adding or removing beliefs, goals or plans
  `.send(maria, tell, colour(box, blue))`. 
Inter-Agent Communication

- agents can communicate using .send(agent, action, content) and .broadcast(action, content)
- possible actions are tell, untell, achieve, unachieve, askOne, askAll, tellHow, untellHow and askHow
- communication triggers events at the receiving agent when adding or removing beliefs, goals or plans

`.send(maria, tell, colour(box, blue)).`

+colour(box, X)
  <- .print("The box is ", X).
Inter-Agent Communication

- agents can communicate using
  \texttt{send(agent, action, content)} and
  \texttt{broadcast(action, content)}
- possible actions are \texttt{tell}, \texttt{untell}, \texttt{achieve}, \texttt{unachieve}, \texttt{askOne}, \texttt{askAll}, \texttt{tellHow}, \texttt{untellHow} and \texttt{askHow}
- communication triggers events at the receiving agent when adding or removing beliefs, goals or plans

\begin{verbatim}
  .send(maria, tell, colour(box, blue)).
  +colour(box, X)
    <- .print("The box is ", X).
  +colour(box, X)[source(Y)] : Y \== self
    <- .print(Y, " told me that the box is ", X).
\end{verbatim}
Annotations

- annotations differentiate beliefs based on their origin or other properties

  \( \text{colour(box1, blue)}[\text{source(bob)}] \).
  \( \sim \text{colour(box1, white)}[\text{source(john)}] \).
  \( \text{colour(box1, red)}[\text{source(percept)}] \).
Annotations

- annotations differentiate beliefs based on their origin or other properties

\[
\text{colour(box1, blue)[source(bob)].} \\
\neg\text{colour(box1, white)[source(john)].} \\
\text{colour(box1, red)[source(percept)].} \\
\text{colourblind(bob)[source(self), degOfCert(0.7)].}
\]
Annotations

- annotations differentiate beliefs based on their origin or other properties
  
  \[
  \text{colour(box1, blue)[source(bob)]}. \\
  \sim\text{colour(box1, white)[source(john)]}. \\
  \text{colour(box1, red)[source(percept)]}. \\
  \text{colourblind(bob)[source(self), degOfCert(0.7)]}. \\
  \]

- they may be nested to indicate belief chains
  
  \[
  \text{loves(maria,bob)[source(john)[source(maria)]]}. \\
  \]
Supermarket Agent

\[ \text{last\_order\_id}(1). \]

\[ +!\text{order}(\text{Product}, \text{Quantity})[\text{source}(\text{Ag})] : \text{true} \]
\[ \text{<- } ?\text{last\_order\_id}(N); \]
\[ \text{OrderId} = N + 1; \]
\[ -+\text{last\_order\_id}(\text{OrderId}); \]
\[ \text{deliver}(\text{Product}, \text{Quantity}); \]
\[ .\text{send}(\text{Ag}, \text{tell}, \]
\[ \text{delivered}(\text{Product}, \text{Quantity}, \text{OrderId})). \]
Introduction

Jason – an Extension of AgentSpeak

Development of Multi-Agent Systems using Jason
  Scenario Analysis
  Agent Design
  Programming

Conclusion

Sources
Scenario Analysis

- identify goals: strategic goals, maintenance goals
- determine tasks to achieve goals
- split tasks into as many small sub-tasks as possible
- assign commitment strategies:
  - blind commitment,
  - single-minded commitment,
  - relativised commitment,
  - open-minded commitment
Agent Design

- follow BDI-model, design common base for all agents
- focus on interaction & communication
- determine collaborative strategies to achieve goals
- assign sub-tasks based on agent type, individual strengths
- promote agents to leaders, design hierarchies
Programming

- implement internal actions for more complex tasks, shared among agents
- use declarative goals instead of procedural ones
  
  \[ +\!g : c <- p ; \ ?g. \]
- differ between global and agent-specific goals
- cover all contexts, fallback plans for error-recovery
Introduction

Jason – an Extension of AgentSpeak

Development of Multi-Agent Systems using Jason

Conclusion

Sources
Conclusion

- agents based on BDI-model, logical reasoning
- (largely) autonomous reactive systems
- focus on collaboration, solving many small tasks to work towards a large goal
- very adaptable, can be deployed to unknown environments
- Jason was developed with BDI concepts in mind
  - allows agent-oriented programming, requires adapted design patterns
Introduction

Jason – an Extension of AgentSpeak

Development of Multi-Agent Systems using Jason

Conclusion

Sources
Sources

