Multi-Agent Programming with Jason

Rafael H. Bordini

University of Durham, U.K.
R.Bordini@durham.ac.uk
http://www.dur.ac.uk/r.bordini

UCPel, 18th of April 2006
Outline

- Introduction
- Overview of AgentSpeak
- Some Features of Jason
- Related Research:
  - Semantics of Communication
  - Plan Exchange
  - Ontological Reasoning
  - Environments, Organisations, and Social Simulation
  - Belief Revision
  - Plan Patterns
- Formal Verification of AgentSpeak Software
  - Model Checking AgentSpeak(F)
  - Property-Based Slicing for AgentSpeak
- Conclusions and Related Work
Part I

Introduction
AgentSpeak is an elegant extension of logic programming for the implementation of BDI agents.

Various extensions were necessary to make it more practical.

*Jason* implements the operational semantics of an extended version of AgentSpeak.

The platform provides a very simple mechanism for defining and running multi-agent systems.

*Jason* is jointly developed with Jomi F. Hübner (FURB, Brazil).
Agent-Oriented Programming

- Put forward by Shoham [JAI, 1993]
- Use of **mentalistc** notions and a **societal** view of computation (anthropomorphism)
- Environment – Agents – Organisations
- Only recently agent programming languages became sufficiently practical
- Most agent development platforms have no formal basis
BDI Architecture

- Intentional Stance (Dennett)
- Practical Reasoning (Bratman)
- IRMA (Bratman, Isreal, Pollack)
- PRS (Georgeff, Lansky)
- dMARS (Kinny)
- BDI Logics and Agent Architecture (Rao, Georgeff)
- Wooldridge, Singh, ...
Generic BDI Architecture

- Sensors
- Input
- BRF
- Beliefs
- Generate
- Options
- Desires
- Filter
- Intentions
- Action
- Effectors
- Output

UCPel, 18/04/06
Part II

Overview of AgentSpeak
AgentSpeak(L)

- Originally proposed by Rao [MAAMAW 1996] as an abstract agent programming language
- Programming language for BDI agents (reactive planning systems)
- Based on PRS and the work on BDI logics
- Influential in the design of other languages
Abstract version of a Mars exploration scenario: a typical day of activity of an autonomous Mars rover

Typical instructions sent to the rover by the ground team:

1. Back up to the rock named Soufflé
2. Place the arm with the spectrometer on the rock
3. Do extensive measurements on the rock surface
4. Perform a long traverse to another rock

It turned out that the robot was not correctly positioned, so scientific data was lost.

Green patches on rocks indicate good science opportunity.

Batteries only work while there is sunlight (“sol” is a Martian day).

Detailed program used in the experiments had 25 plans.
Examples of Plans

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

+!traverse(Coords) :
  safe_path(Coords) <-
  move_towards(Coords).

+!traverse(Coords) :
  not safe_path(Coords) <-
  ...

Multi-Agent Programming  UCPel, 18/04/06
Examples of Plans (II)

```prolog
+!examine(Rock) : 
  correctly_positioned(Rock) <- place_spectrometer(Rock);
  !extensive_measurements(Rock).

+!examine(Rock) :
  not correctly_positioned(Rock) <- !correctly_positioned(Rock);
  !examine(Rock).
```
Part III

Some Features of *Jason*
Annotated predicate:

\[ ps(t_1, \ldots, t_n)[a_1, \ldots, a_m] \]

where \( a_i \) are first-order terms (these have no annotations)

- in the belief base, all predicates have a special annotation
  \[ \text{source}(s_i) \]

where \( s_i \in \{ \text{self}, \text{percept}, id \} \), and \( id \) is any agent label (i.e., name)

Example (belief annotations)

- blue(box1) [source(ag1)].
- red(box1) [source(percept)].
- colourblind(ag1) [source(self), degOfCert(0.7)].
- liar(ag1) [source(self), degOfCert(0.2)].
Plan labels also can have annotations.

Easy to write (in Java) selection functions that use information about the plans contained in such annotations.

Annotation can also be dynamically changed in instances of plans (intentions).

This can be used, e.g., to update the priority that needs to be given to a certain plan.

Example (plan with annotated label)

```java
anotherLabel[chanceOfSuccess(0.7), usualPayoff(0.9), anyOtherProperty] ->
+ b(X) : c(t) <- a(X).
```
Language Extensions (III)

- Strong negation (operator ~)

**Example (strong negation)**

```
+!leave(home)
  : not raining & not ~raining
  <- open(curtains); ...
```

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

- Deletion events used for handling plan failures

**Example (an agent blindly committed to g)**

```
+!g : g <- true.
```

```
+!g : ... <- ... !g.
```

```
-!g : true <- !g.
```
Internal actions can be defined by the user in Java (or other programming languages)

\[ \text{libName.actionName(...) \ldots} \]

Standard (pre-defined) internal actions have an empty library name

Internal action for communication: \(.\text{send}(r, ilf, pc)\)
where \(ilf \in \{\text{tell, untell, achieve, unachieve, tellHow, untellHow, askIf, askOne, askAll, askHow}\}\)

Some other standard internal actions:

- \(.\text{desire}(\text{literal})\)
- \(.\text{intend}(\text{literal})\)
- \(.\text{dropDesires}(\text{literal})\)
- \(.\text{dropIntentions}(\text{literal})\)
- print, sort, list operations, etc.
Green-Patch Plan Revisited

```
+green_patch(Rock) :
  not battery_charge(low) &
  .desire(traverse(C)) <-
    .dropDesires(traverse(C));
  dip.get_coords(Rock, Coords);
  !traverse(Coords);
  !examine(Rock).
```
MAS Configuration File

- **Jason** has a simple language for defining a multi-agent system, where each agent runs its own AgentSpeak interpreter, and an environment can be given by a Java class

```
MAS Auction {

    infrastructure: Saci

    environment: AuctionEnv

    agents: ag1; ag2; ag3;

}
```
System *Architecture* options: Centralised or Saci

- Easy to specify in which *host* agents and the environment will run
  
  ```
  agents:
  agl at host1.dur.ac.uk;
  ```

- Explicitly specifying the file where the agent’s *source code* is to be found
  
  ```
  agents: agl file1;
  ```

- Indicating the *number of instances* of an agent (using the same initial beliefs and plan library)
  
  ```
  agents: agl #10;
  ```
Customising the Infrastructure

- Users can define a specific way the agent interacts with the multi-agent systems infrastructure.

- This is used to customise the way the agent does perception of the environment, receives communication massages, and acts in the environment.

In the configuration file:

```
agents: ag1 agentArchClass MyAgArch;
```

Example of customised architecture class:

```java
import jason.architecture.*;
public class MyAgArch extends AgentArchitecture {
    public void perceive() {
        System.out.println("Getting percepts!");
        super.perceive();
    }
}
```
Customising an Agent Class

This is used to customise the *selection functions* of the AgentSpeak interpreter and other agent-specific functions

- Selection functions
- Belief update and revision
- Functions defining trust/power relations for processing communication messages
- Message and action-feedback (from environment) processing priorities
In actual deployment, there will normally be a real-world environment where the MAS will be situated.

The AgentArchitecture needs to be customised to get perceptions and act on such environment.

We often want a simulated environment (e.g., to test the MAS).

This can be done in Java by extending Jason’s Environment class and using methods such as addPercept(String Agent, Literal Percept).
MAS heathrow {
    environment: HeathrowEnv 
    agents: 
        mds agentClass mds.MDSAgent 
            #5; 
        cph agentArchClass cph.CPHAgArch 
            agentClass cph.CPHAgent 
            #10; 
        bd #3; 
}
Jason Screenshots (II)

Multi-Agent Programming  
UCPel, 18/04/06
Jason is available

Open Source

under GNU LGPL at:

http://jason.sf.net

(kindly hosted by SourceForge)

Jason

by Gustave Moreau (1865)
Oil on canvas, 204 x 115.5 cm.
Musée d’Orsay, Paris.
© Photo RMN. Photograph by Hervé Lewandowski.
Books on Multi-Agent Programming

MULTI-AGENT PROGRAMMING
Languages, Platforms and Applications

Edited by
Rafael H. Bordini
Mehdi Dastani
Jürgen Dix
Amal El Fallah Seghrouchni

Springer
Part IV

Related Research
Speech-Act Based Communication

- Essential for developing multi-agent systems
- Operational semantics for processing messages with the following illocutionary forces:
  - tell / untell (changing beliefs)
  - achieve / unachieve (changing goals)
  - tellHow / untellHow (changing plans)
  - askIf, askAll, askHow

- Joint work with Renata Vieira, Álvaro Moreira, and Mike Wooldridge
Coo-AgentSpeak: recent work on plan exchange

Simple intuition: if you do not know how to do something, ask someone who does

Based on Coo-BDI plan exchange mechanism (Viviana Mascardi, Davide Ancona)

Interesting implementation in Jason (extension/customisation mechanisms suffice)

Joint with Viviana Mascardi, Davide Ancona, and Jomi Hübner
Belief base as a (populated) ontology, whereby:

1. queries to the belief base are more expressive as their results do not rely only on explicitly represented literals but can be inferred from the ontology
2. the notion of belief update is refined given that (ontological) consistency of a belief addition can be checked
3. retrieving a plan for handling an event is more flexible as it is not based solely on unification but on the subsumption relation between concepts
4. agents may share knowledge by using ontology languages such as OWL
Concretely, in *Jason* we use annotations to specify which ontology each belief belongs to.

We use an existing tool (e.g., Racer) to do the ontological reasoning when required.

Increases the need for belief revision.

Joint work with Renata Vieira, Álvaro Moreira, and Jomi Hübner.
Simulated environments should ideally be defined with higher-level languages than Java.

This is particularly useful for social simulation.

Language ELMS defined for this purpose.

Currently being extended with “normative objects”.

Agents can be organised in complex social structures.

Planned integration with Moise+ (Hübner et al.).

Joint work with A.C. Rocha Costa, Fabio Okuyama, and Jomi Hübner.
Belief Revision

- In *Jason* (and other AOP frameworks) consistency of the belief base is left for programmers to ensure.
- Automatic belief revision is typically too expensive.
- Alechina and Logan have recently introduced a new polynomial-time belief-revision algorithm.
- The algorithm makes simplifying assumptions, which nevertheless are realistic for belief bases in AOP.
- Features of *Jason* make it increasingly difficult for programmers to handle belief consistency:
  - belief additions in a plan body ("mental notes")
  - (speech-act based) agent communication
  - derived beliefs from rules of an ontology
- *Jason* will have explicit separation of belief update and revision, both customisable.
Beliefs are annotated with its *dependencies* and its *justifications*

Each way of deriving a belief literal $A$ generates a particular “dependencies” annotation

For example, $(A, [B, C])$, where $A$ is a derived belief and it was asserted by a plan with context $B$ and triggering belief addition $C$ (or derived by an ontology rule $B, C \rightarrow A$)

Each support list has a designated *least preferred* member, to be retracted first if necessary

A (constant-time) function determining least preferred members can be overridden by programmers

A default function gives preference to percepts over communication, and to newer information
Belief Revision – Example

- \( \langle \text{ag1}, \text{tell}, \text{salesUp}(c1) \rangle \)
- \(+\text{salesUp}(C) [\text{source}(A)] : \text{wellManaged}(C) \& \text{trust}(A) \leftarrow +\text{goodToBuy}(C).\)
- \(\text{goodToBuy}(c1)[\text{source}(\text{ag1})]\) is added to the belief base with \([\text{salesUp}(c1), \text{wellManaged}(c1), \text{trust}(\text{ag1})]\) in its “dependencies” list, and \(\text{goodToBuy}(c1)\) is added to the “justifies” lists of the beliefs \(\text{salesUp}(c1), \text{wellManaged}(c1), \text{and trust}(\text{ag1})\)
- From a financial news web service, the agent acquires the belief \(\text{stocks}(c2,10)[\text{source}(\text{percept})]\), and the agent also believes that \(\text{rival}(c2,c1)\)
- \(+\text{stocks}(C,P) : P > 5 \& \text{rival}(C,R) \leftarrow +\sim\text{goodToBuy}(R).\)
- The algorithm would contract \(\text{goodToBuy}(c1)\) because its support is based on communication which is less reliable.
Belief Revision – Example (Cont.)

- The algorithm not only avoids obvious inconsistencies, it saves much work for programmers to keep the belief base updated.
- Suppose the agent receives news that a crooked CEO has just been fired from $c_1$ and has a plan to update its beliefs about $c_1$ being well managed;
- if the *reason-maintenance style* has been chosen, and there is no other justification for $\text{goodToBuy}(c_1)$, then the algorithm would remove not only the $\text{wellManaged}(c_1)$ belief, but also the $\text{goodToBuy}(c_1)$ belief because the latter depends on the former.

Joint work with Natasha Alechina, Brian Logan, Mark Jago, and Jomi Hübner
Patterns of AgentSpeak plans can be used to define various types of declarative goals with sophisticated temporal structures.

Other frameworks extend syntax and semantics for such complex goal constructs.

The use of patterns avoid the need to do so for AgentSpeak and provide the same flexibility as the idea of patterns in object orientation.

Jason to be extended with pre-processing to help automating the generation of plan patterns from higher-level specifications.
Open-minded commitment: a goal is maintained while it has not yet been achieved, is still believed possible to achieve, and still believed necessary.

Let \( f \) be a failure condition and let \( m \) be the motivation.

\[
\text{OMC}_{g,f,m}(P)
\]

\[
\text{BCG}_{g,BDG}(P)
\]

\[
+ f : \text{true} \leftarrow .\text{dropGoal}(g, \text{false}).
\]

\[
- m : \text{true} \leftarrow .\text{dropGoal}(g, \text{true}).
\]

For example:

- drop condition: “no beer at location \((X,Y)\)”
- motivation condition: “my owner wants a beer”
- single known course of action to achieve goal \( l(X,Y) \):

\[
+ ! l(X,Y) : \text{bc}(B) \land B > 0.2 \leftarrow \text{go}(X,Y).
\]
When the pattern $\text{OMC}_l(X,Y), \sim b(X,Y), \sim b$ is applied to the plan above, we get the following program:

- $+!l(X,Y) : l(X,Y) \leftarrow \text{true}$.  
- $+!l(X,Y) : \text{bc}(B) \& B > 0.2 \leftarrow \text{go}(X,Y); \ ?l(X,Y)$.  
- $+!l(X,Y) : \text{true} \leftarrow !l(X,Y)$.  
- $-!l(X,Y) : \text{true} \leftarrow !l(X,Y)$.  
- $+\sim b(X,Y) : \text{true} \leftarrow .\text{dropGoal}(l(X,Y), \text{false})$.  
- $-\text{wb} : \text{true} \leftarrow .\text{dropGoal}(l(X,Y), \text{true})$.

Joint work with Jomi Hübner and Mike Wooldridge
Part V

Model Checking AgentSpeak(F) MAS
AgentSpeak(F): a restricted version of AgentSpeak(L)

CASP (Checking AgentSpeak Programs):
- conversion of specifications written in a simplified BDI logic to LTL
- automatic translation of AgentSpeak(F) into the input language of existing model checkers:
  - PROMELA then using SPIN
  - Java then using JPF2 (Java model checker)

Joint work with Michael Fisher, Willem Visser, and Mike Wooldridge
Main Restrictions

- some disallowed features:
  - uninstantiated variables in triggering events
  - uninstantiated variables in negated literals within a plan’s context (as originally defined by Rao)
  - a predicate symbol used with different arities (SPIN)
  - first order terms (terms can only be constants and variables)

- various translation parameters are required (to define bounds on PROMELA data structures)
Some Features

- Inter-agent communication: `.send(l, ilf, at)`
- Illocutionary forces:
  - `tell`
  - `untell`
  - `achieve`
- Other basic internal actions (printing, arithmetic operations, etc.)
Explicit state, on-the-fly model checker that works directly on Java bytecodes

Checks for deadlocks, assertion violations, and LTL properties

Has been extensively used for finding bugs in large systems

Developed at NASA [Visser et al., ASE’2000]

Now available Open Source:

http://javapathfinder.sf.net/
Java Models of AgentSpeak(F) Agents

- AgentSpeak(F) restrictions are not required
- Much easier to code in than PROMELA, very clear model:
  - Java libraries: (unbound) data structures
  - Instances of objects: set of intentions
  - Easier to implement unification, plan library, etc.
1. \( \text{be} \) is a \textit{wff}; \hspace{1em} (\textit{be}: \text{boolean expression})

2. \( \text{at} \) is a \textit{wff}; \hspace{1em} (\textit{at}: \text{ground atomic formula})

3. \( (\text{Bel} \ l \ \text{at}), (\text{Des} \ l \ \text{at}), \) and \( (\text{Int} \ l \ \text{at}) \) are \textit{wff};

4. \( (\text{Does} \ l \ a) \) is a \textit{wff}; \hspace{1em} (\textit{l}: \text{agent label}, \textit{a}: \text{ground action formula})

5. \( \forall x. (M \ x \ \text{at}) \) and \( \exists x. (M \ x \ \text{at}) \) are \textit{wff}, where \( M \in \{\text{Bel}, \text{Des}, \text{Int}\} \) and \( x \) ranges over a finite set of agent labels;

6. if \( \varphi \) and \( \psi \) are \textit{wff}, so are \( (\neg \varphi), (\varphi \land \psi), (\varphi \lor \psi), (\varphi \Rightarrow \psi), (\varphi \Leftrightarrow \psi) \), always \( (\Box \varphi) \), eventually \( (\Diamond \varphi) \), until \( (\varphi \mathcal{U} \psi) \), and “release”, the dual of until \( (\varphi \mathcal{R} \psi) \);

7. nothing else is a \textit{wff}.
Part VI

Slicing AgentSpeak Programs
Slicing

- Removing parts of a system’s code
- Used in software engineering for various purposes
- Property-based slicing is used in model checking
  - slicing criteria is the property to be verified rather than a variable (as usual in SE)
- State-space reduction techniques are essential for practical model checking
- Property-based slicing is a precise form of under approximation
Slicing Logic Programs

- Our slicing algorithm requires a literal dependence graph (Zhao, Cheng, and Ushijima).
- Originally defined for parallel logic programs (Guarded Horn Clauses).
- Based on two representations of a logic program:
  - **And/Or Parallel Control-Flow Net**: graph where control-flow dependencies are annotated.
  - **Definition-Use Net**: annotations on data dependencies.
- **Literal Dependence Net** (LDN) is then an arc-classified digraph containing all dependencies relevant for slicing a logic program.
- A slice can then be determined by solving a reachability problem in the LDN.
Slicing AgentSpeak Programs

- Using SPIN’s slicer does not work
- Slicing algorithm for AgentSpeak has as input:
  - set of AgentSpeak programs
  - abstract representation of the environment
  - a (BDI) property as the slicing criterion
The algorithm is divided in three stages:

I. the LDN is created:
   - we consider AgentSpeak notation as part of the predicate symbol (except that ![g] in the body of a plan matches +![g] in the triggering events)
   - first create LDNs for each individual agent, then connect them all through actions in plans – environment rules – belief changes

II. mark plans (according to algorithm given next)

III. a slice is obtained by deleting all plans that were not marked is the previous stage
Slicing Algorithm – Stage II

Marking plans given Agents, Environment, LDN, Property

for all subformula \( f \) of Property with Bel, Des, Int, or Does modalities or an AgentSpeak atomic formula do

for all agent \( a \) in the Agents do

for all plan \( p \) in agent \( a \) do

let \( te \) be the node of the LDN that represents the triggering event of \( p \)

if \( f = (\text{Bel } ag \ b) \) then

for all \( b \)-node \( b_i \) labelled \(+b\) or \(-b\) in \( ag \)'s plans, or in the rules in Environment do

if \( b_i \) is reachable from \( te \) in LDN then

mark \( p \)

if \( f = (\text{Des } ag \ g) \) then

for all \( b \)-node \( g_i \) labelled \(!g\) in \( ag \)'s plans do

if \( g_i \) is reachable from \( te \) in LDN then

mark \( p \)
if \( f = (\text{Int } ag \ g) \) then \{note \( t \)-node below, rather than \( b \)-node\}

for all \( t \)-node \( g_i \) labelled \(!g\) in \( ag\)'s plans do

if \( g_i \) is reachable from \( te \) in \( \text{LDN} \) then

mark \( p \)

if \( f = (\text{Does } ag \ a) \) then

for all \( b \)-node \( a_i \) labelled \( a \) in \( ag\)'s plans do

if \( a_i \) is reachable from \( te \) in \( \text{LDN} \) then

mark \( p \)

if \( f \) is an AgentSpeak atomic formula \( b \)
not in the scope of the modalities above

\{meaning \( b \) is true of the \( \text{Environment} \)\} then

for all node \( b_i \) labelled \(+b\) or \(-b\) in the
rules in the \( \text{Environment} \) do

if \( b_i \) is reachable from \( te \) in \( \text{LDN} \) then

mark \( p \)
Why is the State Space Reduced?

Reduction can happen for two reasons:

1. By removing plans that would increase the length of a computation for an agent to handle particular events (i.e., an intention) before the truth of the property can be determined.

2. When all the plans that are used to handle particular external events can be removed: at any point during an intention execution there can be reachable states in which other intentions (other focuses of attention) are created to handle (irrelevant) events; this type of slicing eliminates all such branches of the computation tree.
Examples of Specifications

(1)

\[ \Box((\text{Does amr place\_spectrometer(R)})) \rightarrow (\text{Bel amr correctly\_positioned(R)}) \]

(2)

\[ \Box((\text{Int amr transmit\_remaining\_data(Day)})) \rightarrow \Diamond \neg ((\text{Bel amr data(spect,Rock,Day,\_)})) \land \neg (\text{Bel amr downlink(ground,spect,Rock,Day)})) \]
Results

For property (1), plans $c_1 - c_4$ (downlink) are excluded from the slice (out of 25 plans)

For property (2), plans $r_3$ (reacting to ordinary possible target rocks) is excluded

Experiments were run on a machine with an MP 2000+ (1666 MHz) processor with 256K cache and 2GB of RAM (266 MHz):

- Specification (1)
  - **Before Slicing**: SPIN used 606MB of memory ($1.18 \times 10^6$ states) and took 86s
  - **After Slicing**: down to 407MB (945,165 states) and 64s
  - **Reduction**: 33% (memory), 25.6% (time)

- Specification (2)
  - **Before Slicing**: 938MB of memory ($2.87 \times 10^6$ states) and took 218s
  - **After Slicing**: down to 746MB ($2.12 \times 10^6$ states) and 162s
  - **Reduction**: 21% (memory), 26% (time)
Part VII

Conclusions and Related Work
Conclusions

- A practical approach for programming and verifying multi-agent systems:
  - programmed in a BDI logic programming language
  - properties specified in a (simplified) BDI logic
- Reduction to standard LTL model-checking allows the use of existing (sophisticated) model-checkers
- Slicing can significantly reduce the state space of an AgentSpeak system
- Various ongoing research projects aiming at improving various aspects of multi-agent programming (made practical in the context of Jason)
Other Work on Agent-Oriented Programming

- 3APL (Dastani, van Riemsdijk, Meyer, ...)
- MetateM (Michael Fisher, Chiara Ghidini, Benjamin Hirsch)
- ConGoLog (Lesperance, Levesque, ... / Boutilier – DTGolog)
- Teamcore/MTDP (Milind Tambe, ...)
- IMPACT (Subrahmanian, Kraus, Dix, Eiter)
- CLAIM (Amal El Fallah-Seghrouchni, ...)
- Minerva (Leite, ...)
- STAPLE (Kumar, Cohen, Huber)
- Go! (Clark, McCabe)
- Jadex (Braubach, Pokahr), Jack (AOS), etc.
Other Work on Model Checking Agents

- Lomuscio and Penczek (OBDD and SAT-based model checking for epistemic and deontic logics)
- Ryan and Schobbens (model checking and refinement in ATL)
- Wooldridge and van der Hoek (model checking for ATEL)
- van der Meyden (MCK model checker for an epistemic logic)
- Benerecetti (model checking for BDI logics)
- Cimatti, Singh, Pacheur, ...