A Verifiable Approach to Programming Multi-Agent Systems

Rafael H. Bordini

Department of Computer Science
University of Durham, Durham, U.K.

R.Bordini@durham.ac.uk
http://www.dur.ac.uk/r.bordini
Outline

- Introduction
- Overview of AgentSpeak
- Main Features of *Jason*
- Model Checking AgentSpeak(F) Multi-Agent Systems
- Slicing AgentSpeak Programs
- Conclusions and Ongoing Work
Introduction

- AgentSpeak is a thoughtful and elegant extension of logic programming for the implementation of reactive planning systems
- Various extensions were necessary to make it more practical (Jason)
- Model checking techniques for MAS are very recent
- Verification of multi-agent systems written in AgentSpeak using existing model checkers
- Model checking programs rather than designs!
Agent-Oriented Programming

- Proposed by Shoham [JAI, 1993]
- Use of mentalistic notions and a societal view of computation (anthropomorphism)
- Environment – Agents – Organisations
- Programming languages for agents are still incipient
- Most agent 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)
Generic BDI Architecture

- **Sensors**
  - Input
  - BRF
  - Beliefs
  - Generate Options
- **Desires**
  - Filter
  - Intentions
  - Action
  - Effectors
  - Output
Overview of AgentSpeak
AgentSpeak(L)

- Originally proposed by Rao [MAAMAW 1996]
- Programming language for BDI agents (reactive planning systems)
- Natural extension of logic programming
- Influential in the design of other languages
- Practical extensions and working interpreter (*Jason*)
- Operational semantics / definition of BDI modalities
Example

\[
\text{likes(joao\_gilberto).}
\]

\[
\text{+concert(A,V) : likes(A)}
\]
\[
\text{<- !book\_tickets(A,V).}
\]

\[
\text{+!book\_tickets(A,V) : not busy(phone)}
\]
\[
\text{<- ?phone\_number(V,N);} \quad \text{call(N);} \quad \text{...;}
\]
\[
\text{!choose\_seats(A,V).}
\]
Scenario for a Running Example

- 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) <-
        ...

11/10/04 Seminar – University of Namur
Examples of Plans (II)

+ !examine(Rock) :
   correctly_positioned(Rock) ←
   place_spectrometer(Rock);
   !extensive_measurements(Rock).

+ !examine(Rock) :
   not correctly_positioned(Rock) ←
   !correctly_positioned(Rock);
   !examine(Rock).
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 (and respective replies)

- Joint work with Álvaro Moreira and Renata Vieira
Plan Exchange

- Coo-AgentSpeak: recent work on plan exchange
- Simple intuition: if you do not know what to do, ask someone who does
- Based on Coo-BDI plan exchange mechanism (Viviana Mascardi, Davide Ancona)
- Interesting implementation in *Jason* (extention mechanisms)
- Joint with Viviana Mascardi, Davide Ancona, and Jomi Hübner
Preliminaries (Operational Semantics)

- From the SOS of AgentSpeak(L):
  - Agent: \( ag = \{bs, ps\} \)
  - Circumstance \( C \) is a tuple \( \langle I, E, A \rangle \)
    - set of intentions \( I \) (an intention is a stack of partially instantiated plans)
    - set of events \( E \) (an event is a pair \( \langle te, i \rangle \), where \( te \) is a triggering event and \( i \) is an intention)
    - the set \( A \) of actions (to be performed)
  - The semantic rules define a transition relation
    \[ \langle ag, C \rangle \xrightarrow{} \langle ag', C' \rangle \]
  - \( C_I \) refers to component \( I \) of \( C \) (similarly for others)
  - The intention with plan \( p \) on top of intention \( i \) is denoted by \( i[p] \)
Definition 1 (Belief in AgentSpeak(L) agents) We say that an AgentSpeak(L) agent $ag$, regardless of its circumstance $C$, believes a formula $\varphi$ iff it is included in the agent's belief base; that is, for an agent $ag = \langle bs, ps \rangle$:

$$\text{BEL}_{\langle ag, C \rangle}(\varphi) \equiv bs \models \varphi$$
Intention (1)

- Auxiliary function $agls : \mathcal{I} \rightarrow \mathcal{P}(\Phi)$

- $\mathcal{I}$ is the domain of all individual intentions (a stack of partially instantiated plans)

- Returns all achievement goals in the triggering event part of the plans

- For any $i \in \mathcal{I}$:

  $$agls(T) = \{\}$$

  $$agls(i[p]) = \begin{cases} 
  \{at\} \cup agls(i) & \text{if } p = +!at : ct \leftarrow h \\
  agls(i) & \text{otherwise}
  \end{cases}$$
Definition 2 (Intention in AgentSpeak(L) agents) An AgentSpeak(L) agent $ag$ intends $\varphi$ at circumstance $C$ iff it has $\varphi$ as an achievement goal that currently appears in its set of intentions $I$, or $\varphi$ is an achievement goal that appears in the (suspended) intentions associated with events in $E$. For an agent $ag$ and its circumstance $C$, we have:

\[
\text{INTEND}_{\langle ag,C\rangle}(\varphi) \equiv \varphi \in \bigcup_{i \in C_I} \text{agls}(i) \lor \varphi \in \bigcup_{(te,i) \in C_E} \text{agls}(i)
\]
Definition 3 (Desire in AgentSpeak(L) agents)  

An agent at circumstance $C$ desires a formula $\varphi$ iff $\varphi$ is an achievement goal in $C$’s set of events $E$ (associated with any intention $i$), or $\varphi$ is a current intention of the agent; more formally:

$$\text{DES}_{ag,C}(\varphi) \equiv \langle +!\varphi, i \rangle \in C_E \lor \text{INTEND}_{ag,C}(\varphi)$$
Main Features of Jason
Jason

- An interpreter for an extended version of AgentSpeak
- Distribution over the net using SACI (Jomi Hübner)
- Implements the operational semantics of AgentSpeak
- Jointly developed with Jomi Hübner (and others)

Some of its features are:

- strong negation, so both closed-world assumption and open-world are available;
- handling of plan failures;
- speech-act based inter-agent communication (and annotation of beliefs with information sources);
Jason (II)

Features (continued):

- annotations on plan labels, which can be used by elaborate (e.g., decision-theoretic) selection functions;
- support for developing Environments (in Java);
- running multi-agent systems on a network (using SACI);
- fully customisable (in Java) selection functions, trust functions, and overall agent architecture (perception, belief-revision, inter-agent communication, and acting);
- a library of essential “internal actions”;
- straightforward extensibility by user-defined internal actions, programmed in Java.
Language Extensions

- Internal action for communication: `.send(r, tellHow, P)`

- Annotated predicate:

  $ps(t_1, \ldots, t_n)[a_1, \ldots, a_m]
  
  where $a_i \in \{\text{self, percept, id}\}, i \leq m$, $id$ is an agent label

- Plan labels can have annotations: a list of ground terms that can be dynamically changed in instances of plans (intentions)

- Example of a plan with annotated label:

  anotherLabel[chanceSuccess(0.7), usualPayoff(0.9), anyOtherProperty] $\rightarrow$ 
  $+b(X) : c(t) \leftarrow a(X)$. 

Language Extensions (II)

- Strong negation (\neg operator)
- Deletion events used for handling plan failures
- Some other standard internal actions:
  - \texttt{.desire(literal)}
  - \texttt{.intend(literal)}
  - \texttt{.dropDesires(literal)}
  - \texttt{.dropIntentions(literal)}
  - printing, arithmetic/relational expressions, etc.
Green-Patch Plan Revisited

\begin{verbatim}
+green_patch(Rock) :

  not battery_charge(low) & desire(traverse(C)) <-
                      .dropDesires(traverse(C));
  dip.get_coords(Rock, Coords);
  !traverse(Coords);
  !examine(Rock).
\end{verbatim}
MAS Configuration File

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

```java
MAS Auction {
    architecture: Saci
    environment: AuctionEnv
    agents: ag1; ag2; ag3;
}
```
MAS Configuration (II)

- System Architecture options: Centralised or Saci

- Easy to specify in which host agents and the environment will run

  agents:
  ag1 at host1.dur.ac.uk;

- Explicitly specifying the file where the agent’s source code is to be found

  agents: ag1 file1;

- Indicating the number of instances of an agent (using the same initial beliefs and plan library)

  agents: ag1 #10;
Customising an Overall Agent Architecture

- Users can define a specific overall (rather than reasoning) architecture for an agent
- This is used to customise the way the agent does perception of the environment, receives communication messages, does belief revision, and acts in the environment

In the configuration file:

```
agents: ag1 agentArchitecture MyArch;
```

Example of customised architecture class:

```
import jason.*;
public class MyAgArch extends CentralisedAgArch {
    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
  - Event selection function
  - Applicable plans selection function
  - Intention selection function
- Functions defining trust/power relations for processing communication messages
- Message and action-feedback (from environment) processing priorities
Customising an Agent Class (II)

In the configuration file:

```plaintext
agents: ag1 agentClass MyAgClass;
```

Example of customised agent class:

```java
import jason.*;
import java.util.*;
public class MyAgClass extends Agent {

    public Event selectEvent(List evList) {
        System.out.println("Selecting an event from "+evList);
        return((Event)evList.remove(0));
    }
}
```
import java.util.*;
import jason.*;
public class auctionEnv extends Environment {

    public auctionEnv() {
        getPercepts().add(......);
    }

    public boolean executeAction(String ag, Term action) {
        if (action.hasActionSymb("place_bid")) {
            Integer x = new Integer(action.parameter(2).toString());
            bigs.put(ag, x);
        }

        ...

        // determines the winner
        getPercepts().add(winner);
        return true;
    }

}
Jason is available

Open Source

under GNU LGPL at:

http://jason.sourceforge.net

(kindly hosted by SourceForge)
Model Checking AgentSpeak(F) Multi-Agent Systems
Model Checking AgentSpeak(F)

- **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: \texttt{.send(l, ilf, at)}

- Illocutionary forces:
  - \texttt{tell}
  - \texttt{untell}
  - \texttt{achieve}

- Other basic internal actions (printing, arithmetic operations, etc.)
JPF2

• Explicit state, on-the-fly model checker that works directly on Java bytecodes
• Checks for deadlock, assertion violations, and LTL properties
• Has been extensively used for finding bugs in large systems
• Developed at NASA [Visser et al., ASE’2000]
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.
- Fairness constraint alleviates state explosion:
  - One reasoning cycle per agent in a fixed order
Property Specification Language

1. \( be \) is a \textit{wff}; \hspace{1cm} (\textit{be}: \text{boolean expression})
2. \( at \) is a \textit{wff}; \hspace{1cm} (\textit{at}: \text{ground atomic formula})
3. \((\text{Bel} \ l \ at), (\text{Des} \ l \ at), \text{and} (\text{Int} \ l \ at)\) are \textit{wff};
4. \( \forall x. (M \ x \ at) \) and \( \exists x. (M \ x \ at) \) are \textit{wff}, where \( M \in \{\text{Bel}, \text{Des}, \text{Int}\} \) and \( x \) ranges over a finite set of agent labels;
5. \((\text{Does} \ l \ a)\) is a \textit{wff}; \hspace{1cm} (\textit{l}: \text{agent label, a: ground action formula})
6. if \( \varphi \) and \( \psi \) are \textit{wff}, so are \( (\neg \varphi), (\varphi \land \psi), (\varphi \lor \psi), (\varphi \Rightarrow \psi), (\varphi \Leftrightarrow \psi) \), \( (\varphi \Leftrightarrow \psi) \), \text{always} \( (\Box \varphi) \), \text{eventually} \( (\Diamond \varphi) \), \text{until} \( (\varphi \mathcal{U} \psi) \), and \text{“release”}, the dual of until \( (\varphi \mathcal{R} \psi) \);
7. nothing else is a \textit{wff}. 
Preliminary Results

Results of checking one of the specification of the garbage collecting robot scenario:

- SPIN: 333,413 states, used 210.51 MB of memory, and took nearly 65.78 seconds to complete.
- JPF2: 236,946 states, used 366.68 MB of memory, and took 18:49:16 hours to complete!

In another setting of the scenario, where garbage is placed at fixed positions, the verification took JPF2 76.63 seconds to finish, and 5.25 seconds for SPIN.
Brief Comparison

- SPIN seems to scale better than JPF2
- Using the current (preliminary) versions of the PROMELA and Java models
- Java is widely used in the implementation of MAS, and provides clear and easily extensible models of AgentSpeak(F)
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)
- Abstraction 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 AgentSpeak Slicing Algorithm

- 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 \(\text{Agents, Environment, LDN, Property}\)

\[
\text{for all subformula } f \text{ of } \text{Property} \text{ with Bel, Des, Int, or Does modalities or an AgentSpeak atomic formula do}
\]

\[
\text{for all agent } ag \text{ in the } \text{Agents do}
\]

\[
\text{for all plan } p \text{ in agent } ag \text{ do}
\]

\[
\text{let } te \text{ be the node of the LDN that represents the triggering event of } p
\]

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

\[
\text{for all } b\text{-node } b_i \text{ labelled } +b \text{ or } -b \text{ in } ag\text{'s plans, or in the facts and right-hand side of rules in Environment do}
\]

\[
\text{if } b_i \text{ is reachable from } te \text{ in } \text{LDN then}
\]

\[
\text{mark } p
\]

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

\[
\text{for all } b\text{-node } g_i \text{ labelled } !g \text{ in } ag\text{'s plans do}
\]

\[
\text{if } g_i \text{ is reachable from } te \text{ in } \text{LDN then}
\]

\[
\text{mark } p
\]
Slicing Algorithm – Stage II (Cont.)

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 \( Environment \)\} then
   for all node \( b_i \) labelled \( +b \) or \( -b \) in the facts and 
   right-hand side of rules in the \( Environment \) do
     if \( b_i \) is reachable from \( te \) in \( \text{LDN} \) then
       mark \( p \)
Definition 4 (Slicing Algorithm Correctness) A slicing algorithm for AgentSpeak $\sigma$ is correct if for any finite set of AgentSpeak programs $A$, abstract environment $E$, and property $P$, for $A'$ such that $\sigma(A, E, P) = A'$, $A, E \models P$ if and only if $A', E \models P$.

Lemma 1 (Belief subformula) Truth of a formula $(\text{Bel} ag b)$ is affected: (i) when $+b$ or $-b$ appears in an executing plan, or (ii) by belief revision from perception of the environment. Any plan in the system can make either (i) or (ii) happen if, and only if, it is marked by stage II of the algorithm whenever $(\text{Bel} ag b)$ is a subformula of property $P$.

Theorem 1 The slicing algorithm for AgentSpeak introduced in this paper is correct in the sense of Definition 1.

Proof. By structural induction on the wff of the property specification language, using five lemmas referring to the base cases.
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)
\[ \square((\text{Does } amr \text{ place}_{-}\text{spectrometer}(R)) \rightarrow (\text{Bel } amr \text{ correctly}_{-}\text{positioned}(R))) \]

(2)
\[ \square((\text{Int } amr \text{ transmit}_{-}\text{remaining}_{-}\text{data}(\text{Day})) \rightarrow (\lozenge \neg((\text{Bel } amr \text{ data}(\text{spect},\text{Rock},\text{Day},_))) \land (\neg(\text{Bel } amr \text{ downlink}(\text{ground},\text{spect},\text{Rock},\text{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)
Conclusions and Ongoing 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
- Initial results indicate that:
  - while Java provides a more appropriate target language than PROMELA, JPF2 does not (currently) scale as well as SPIN
  - slicing can significantly reduce the state space of MAS: an important impact on practical agent verification
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, ...)
- 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 Mayden (MCK model checker for an epistemic logic)
- Benerecetti (model checking for BDI logics)
- Cimatti, Singh, Pacheur, ...
Ongoing and Future Work (Programming)

- Use of ontologies as underlying structure of the belief base (Semantic Web)
- Social structure and regulation: organisations, values, norms
- More realistic applications
- E-Science applications (Social Simulations on the Grid)
Ongoing and Future Work (Verification)

- Other techniques for coping with the state explosion problem
- Combining with deductive verification
- Proving correctness of the AgentSpeak(F) translation and finalise the one for the slicing algorithm
- Improving the models generated from translations
- Further comparison between various target model-checkers
- Other experiments (autonomous spacecraft control)