

# Model Checking and Abstraction Techniques for Agent Verification

Rafael H. Bordini<sup>1</sup>

*joint work with:*

Michael Fisher<sup>2</sup>, Willem Visser<sup>3</sup>, Michael Wooldridge<sup>2</sup>

R.Bordini@durham.ac.uk, M.Fisher@csc.liv.ac.uk,  
wvisser@email.arc.nasa.gov, M.J.Wooldridge@csc.liv.ac.uk

<sup>1</sup> University of Durham, U.K.

<sup>2</sup> University of Liverpool, U.K.

<sup>3</sup> RIACS/NASA Ames Research Center, U.S.A.



# Outline

- Introduction
- Overview of AgentSpeak
- Model Checking AgentSpeak(F) Multi-Agent Systems
  - AgentSpeak(F) Restrictions and Features
  - Property Specification Language
- SPIN × JPF2
- Slicing AgentSpeak Programs
- Case Study (Autonomous Mars Rover)
- Conclusions and Future Work

# Introduction

- AgentSpeak is a thoughtful and elegant extension of logic programming for the implementation of reactive planning systems
- 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!

# AgentSpeak(L)

- Originally proposed by Rao [MAAMAW 1996]
- Programming language for BDI agents
- Natural extension of logic programming
- Practical extensions and working interpreter (**Jason**)
- Operational semantics / definition of BDI modalities

# Example

```
likes(joao_gilberto).
```

```
+concert(A,V) : likes(A)
```

```
  <- !book_tickets(A,V).
```

```
+!book_tickets(A,V) : not(busy(phone))
```

```
  <- call(V);
```

```
    ...;
```

```
    !choose_seats(A,V).
```



# Model Checking AgentSpeak(F)

- AgentSpeak(F): a restricted version of AgentSpeak(L)
- CASP (Checking AgentSpeak Programs) [CAV03]:
  - 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 [AAMAS 2003]
    - Java then using JPF2 (Java model checker) [ProMAS 2003]

# 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.)

# 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 *wff*; ( $be$ : boolean expression)
2.  $at$  is a *wff*; ( $at$ : ground atomic formula)
3.  $(\mathbf{Bel} \ l \ at)$ ,  $(\mathbf{Des} \ l \ at)$ , and  $(\mathbf{Int} \ l \ at)$  are *wff*;
4.  $\forall x.(M \ x \ at)$  and  $\exists x.(M \ x \ at)$  are *wff*, where  
 $M \in \{\mathbf{Bel}, \mathbf{Des}, \mathbf{Int}\}$  and  $x$  ranges over a finite set of agent labels;
5.  $(\mathbf{Does} \ l \ a)$  is a *wff*; ( $l$ : agent label,  $a$ : ground action formula)
6. if  $\varphi$  and  $\psi$  are *wff*, so are  $(\neg\varphi)$ ,  $(\varphi \wedge \psi)$ ,  $(\varphi \vee \psi)$ ,  $(\varphi \Rightarrow \psi)$ ,  
 $(\varphi \Leftrightarrow \psi)$ , 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 *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

- 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 [AAMAS04] 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 a  $+!g$  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 (see algorithm)
  - III a slice is obtained by deleting all plans that were *not* marked in 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  $ag$  in the Agents do
    for all plan  $p$  in agent  $ag$  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
        facts and right-hand side of 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  $p$ 
```

# 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 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 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 LDN then  
      mark  $p$ 
```

# Correctness (Outline)

**Definition 1 (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)** *A formula  $(Bel\ ag\ b)$  can only become true in regard to an AgentSpeak agent  $ag$  under two circumstances: (i) when  $+b$  appears in the body of one of the agent's plans, or (ii) by belief revision from the agent's 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  $(Bel\ ag\ b)$  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 logic used to write the specifications, using the five lemmas that refer to the base cases.

# Case Study

- Abstract version of a Mars exploration scenario: a typical day of activity of an autonomous 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
- 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) <-  
    ...
```

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

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

# Why the State Space is Reduced?

- Reduction can happen for two reasons:
  - 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
  - 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)

$$\square((\text{Does } amr \text{ place\_spectrometer}(R)) \rightarrow (\text{Bel } amr \text{ correctly\_positioned}(R)))$$

(2)

$$\square((\text{Int } amr \text{ transmit\_remaining\_data}(\text{Day})) \rightarrow \diamond\neg((\text{Bel } amr \text{ data}(\text{spect}, \text{Rock}, \text{Day}, \_)) \wedge \neg(\text{Bel } amr \text{ downlink}(\text{ground}, \text{spect}, \text{Rock}, \text{Day}))))$$

# Results

- For property (1), plans  $c1-c4$  (downlink) are *excluded* from the slice
- For property (2), plans  $r3$  (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

- A practical approach to the verification of 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

# Ongoing and Future Work

- 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
- More realistic applications