Exploring AOP from an OOP Perspective

REM W. COLLIER, SEÁN RUSSELL AND DAVID LILLIS

SCHOOL OF COMPUTER SCIENCE, UNIVERSITY COLLEGE DUBLIN, IRELAND

Background

Masters in Advanced Software Engineering offered in UCD since 2005

A module entitled Agent Oriented Software Engineering is offered.

Students are typically professionals with 5+ years of software engineering experience.

Students exposed to a variation of AgentSpeak(L)

Promote agent abstraction to focus on coordinating multiple decentralised problem-solvers, to provide intelligent decision-making abilities.

Informal feedback:

Would you consider using this in an industrial setting?

No!

Motivation

1. How can we map the **concepts of OOP** to the **concepts of AOP** to make it easier for programmers to bridge the cognitive gap to a new paradigm?

2. How can we use these insights to help the **development of agent programming languages** so that they are attractive to OOP programmers?

AgentSpeak(L)

AgentSpeak(L) can be prosaically described as an **event-driven language**:

- **Plan Rules** are fired based on both a triggering event and some context. Managed by the manipulation of **intentions**.
- Program state is modeled as a set of **beliefs**, (realised as atomic predicate logic formulae)
- **Events** - either external (environment-based) or internal (**goal**-based) - are generated and added to an event queue.
- Events are **removed from the event queue** and **matched to some event handler,** which is then fired.
- The **matching process** checks both that the **event handler applies to the event** and that the handler can be executed based on the **context**, which defines valid program states in which the handler may be fired.

AgentSpeak(L) and OOP

OOP	AOP	Motivation for mapping
Fields	Beliefs	Object/Agent state.
Methods	Plan Rules	Behaviour definition.
Method Calls	Goals	Behaviour calling.
Messages	Events	Behaviour triggering.
Threads	Intentions	Behaviour execution.

Beliefs equivalent to **Fields**

In OOP, the **state** of an object is defined as a set of **fields** that hold values.

In AgentSpeak(L), state is a set of **beliefs** that define relations on values.

Two operations commonly associated with fields are:

Assignment: e.g. `value = 5;` or `name = "Rem";`

Comparison: e.g. `value == 3` or `name.equals("Fred")`

Beliefs equivalent to **Fields**

AgentSpeak(L) offers equivalents to these operations:

Assignment: this involves first dropping the existing belief and then adopting a new belief with the new value: `-value(0); +value(5)` or `-name("Bob"); +name("Rem")`.

Comparison: this can be done in two places – the plan rule context or via a query statement: `<tr> : name("Fred") <- …` or `?name("Fred")`.

While fields can be mapped onto beliefs, one key piece of information is lost: **types**.

Plan Rules equivalent to **Methods**

These concepts refer to the definition of **behaviours**:

- In OOP, behaviour is defined by **methods**.
- In AgentSpeak(L), behaviour is defined by **plan rules**.

Both forms are essentially **labelled blocks of procedural code** that can be executed on demand.

- In OOP, a method is executed when its signature is matched to a message received by the object.
- In AgentSpeak(L), a plan rule is executed whenever an event matching its triggering event is processed and the rule's context is satisfied.

Plan Rules equivalent to Methods

An assumption:

> *If algorithms are a typical way for defining behaviour in OOP and methods are the common mechanism for implementing algorithms, would it not be natural for somebody learning AgentSpeak(L) to attempt to implement some established algorithms using the language?*

For example, what about the selection sort algorithm?

```
Algorithm SelectionSort(A, n):
  for j = 0 to n-1 do
    minIndex = j
    for k = j+1 to n-1 do
      if (A[minIndex] < A[k]) then
        minIndex = k
    if (minIndex <> j) then
      temp = A[j]
      A[j] = A[j+1]
      A[j+1] = temp
  return A
```

```
!do_sort([7, 5, 12, 15, 3]);

+!do_sort(L) <-
    _size(L, S);
    !outerLoop(L, S, 0);
    ?sorted(L2);
    _print(L2).

+!outerLoop(L, S, X) <-
    +min_index(X);
    !innerLoop(L, S, X);
    ?min_index(Z);
    -min_index(Z);
    !update(L, S, X, Z).

+!update(L, S, X, Z) : X < Z <-
    _swap(L, X, Z, L2);
    !outerLoop(L2, S, X+1).

+!update(L, S, X, Z) <-
    !outerLoop(L, S, X+1).
```

```
+!outerLoop(L, S, X) <-
     +sorted(L).

+!innerLoop(L, S, X) : X < S <-
    _elementAt(L, X, T);
    !compare(L, X, T);
    !innerLoop(L, S, X+1).

+!innerLoop(L, S, X) <-
     _skip().

+!compare(L,X,T):min_index(Y)<-
    _elementAt(L, Y, S);
    !compare(L, X, Y, S, T).

+!compare(L, X, Y, S, T) : S < T <-
    -min_index(Y);
    +min_index(X).

+!compare(L, X, Y, S, T) <-
     _skip().
```

Increased Complexity:

- This can result in **loss of readability**.

Rule Explosion:

- one method has been mapped to 10 rules.
- Separate rules for true and false cases in conditionals and loops.

Return Values:

- sub-goal calls do not return values.
- Value must be stored as a belief (in global state) and queried afterwards.

Events equivalent to **Messages**

From an OOP perspective, an object receives a message, and then invokes a method by matching the message to a method signature; searching up the inheritance tree as necessary.

In AgentSpeak(L) it is the event that is matched against a plan rule.

Two types of events possible:

- **Goal events** are raised whenever a goal is adopted or retracted, which can only be done by the agent itself.
- **Belief events** are raised whenever a belief is adopted or retracted, and can be done by the agent itself but also in response to external factors (e.g. environment, communication with other agents).

Intentions equivalent to **Threads**

At a high level, intentions represent the agent's efforts at achieving its goals.

- An agent creates a new intention for every external event that it matches to a plan rule.

In situations where an agent has multiple intentions, intention execution is **interleaved**.

- On each iteration, one intention is selected and executed.

Clearly - intentions operate in a similar way to threads.

But does AgentSpeak(L) reflect this? NO!

- No return values: use of global state to store returned value

Solutions:

1. Introduce support for mutual exclusion
2. Allow goals to return values
3. Support local variables

Introduction to ASTRA

The ASTRA Language is an attempt to address some shortcomings of classic AgentSpeak(L).

Specifically, ASTRA:

- supports typed variables (closely tied to Java's type system)
- includes support for local variables and assignment
- provides and extended suite of statement types
- allows return values
- supports multiple critical areas
- introduces additional event types
- provides a cleaner model for defining internal actions
- engenders multiple inheritance based reuse through a multi-agent level type system
- adopts Java syntactic sugar to promote familiarity

ASTRA and Extended Plan Syntax

Core programming constructs in ASTRA include:

- **if** statement the most basic form of flow control
- **while** loop usual method of repetition in programming
- **foreach** loop repeats the same actions for every matching binding of a formula
- **try ... recover** allows for the recovery from failed actions
- **Local variable** declaration declares a variable for use within a plan rule
- **assignment** allows the value of a local variable to be changed
- **Query** binds the values of beliefs to variables
- **wait** pauses execution until condition if true
- **when** performs block of code when condition is true
- **send** sends message to another agent
- **synchronized** enables mutual exclusion in critical sections

More Information

Website: http://astralanguage.com

- Cookbook (~50 example programs)
- User Guide (slowly improving)
- API Reference Guide (info on some of the standard modules provided)

Eclipse Plugin: http://astralanguage.com/update

Email Contact: rem.collier@ucd.ie

ASTRA and Extended Plan Syntax

```
rule +!sort(list L, list R) {
    R = L;
    int j = 0;
    while (j < P.size(R)) {
        int min = j;
        int k = j+1;
        while (k < P.size(R)) {
            if (P.valueAsInt(R, min) > P.valueAsInt(R, k))
                min = k;
            k++;
        }

        if (min ~= j) {
            R = P.swap(R, min, j);
        }
        j++;
    }
}
```

ASTRA and Mutual Exclusion

```
agent Racy {
    module Console C;
    initial ct(0);
    initial !init(), !init();

    rule +!init() {
        query(ct(int X));
        +ct(X+1);
        -ct(X);
    }

    rule +ct(int X) {
        C.println("X = " + X);
    }
}
```

```
agent Racy {
    module Console C;
    initial ct(0);
    initial !init(), !init();

    rule +!init() {
        synchronized (ct_tok) {
            query(ct(int X));
            +ct(X+1);
            -ct(X);
        }
    }

    rule +ct(int X) {
        C.println("X = " + X);
    }
}
```

ASTRA and Internal Actions

ASTRA links the agent and Java layers through the use of modules (implemented as Java classes).

Methods in the module can be annotated, exposing themselves to the agent layer in different ways:

- **@ACTION**: these methods are internal actions. They return a Boolean value indicating the success or failure of the action.
- **@TERM**: these methods represent basic calculations or return data from some underlying model in a form that can be represented as a term in the language (e.g. an int, float, string, list or object).
- **@FORMULA**: these methods are constructors that return logical formula instances in ASTRA (ranging from Boolean values to logical formulae that can be matched against the beliefs of the agent).
- **@SENSOR**: these methods generate beliefs that are added to the agent's state.

ASTRA and Internal Actions

```
package ex;

import astra.core.Module;

public class MyModule
        extends Module {

    @TERM
    public int max(int a, int b){
        return Math.max(a, b);
    }

    @ACTION
    public boolean printN(int n){
        System.out.println(n);
        return true;
    }
}
```

```
package ex;

agent Bigger {
    module MyModule m;

    initial num(45, 67);
    initial !init();

    rule +!init() {
        query(num(int X, int Y));
        int n = m.max(X,Y);
        m.printN(n);
    }
}
```