Reflecting on Agent Programming with AgentSpeak(L)

Rem W. Collier, Seán Russell, and David Lillis

School of Computer Science,
University College Dublin, Ireland
{rem.collier,sean.russell,david.lillis}@ucd.ie

Abstract. Agent-Oriented Programming (AOP) researchers have successfully developed a range of agent programming languages that bridge the gap between theory and practice. Unfortunately, despite the in-community success of these languages, they have proven less compelling to the wider software engineering community. One of the main problems facing AOP language developers is the need to bridge the cognitive gap that exists between the concepts underpinning mainstream languages and those underpinning AOP. In this paper, we attempt to build such a bridge through a conceptual mapping that we subsequently use to drive the design of a new programming language entitled ASTRA, which has been evaluated by a group of experienced software engineers attending an Agent-Oriented Software Engineering Masters course.

Keywords: Agent-Oriented Programming, AgentSpeak(L), ASTRA

1 Introduction

The Agent-Oriented Programming (AOP) paradigm is nearly 25 years old. Since its inception, a number of established AOP languages have emerged, with the most prominent being: 2/3APL [1, 2], GOAL [3] and Jason [4]. However, while these languages have received much critical success within the AOP community, they have been less well received by the wider software engineering community.

A useful barometer for the view of this wider community has been the students enrolled on an Agent-Oriented Software Engineering (AOSE) module that is part of a Masters in Advanced Software Engineering offered at University College Dublin since 2005. Students on this course typically have 5 or more years of industrial software engineering experience and are senior software engineers in their respective companies. During the course, the students are exposed to an AgentSpeak(L)-based language, which has been one of AF-AgentSpeak [5], Jason [4], and our most recent agent-programming language, ASTRA [6].

Each year, the students have been asked to provide informal feedback on the AOP language(s) used and to comment on whether they would consider using such a language in a live industry application. The common response has been "no", with typical criticisms being the lack of tool support and the perceived learning curve required to master an AOP language.

The lack of tool support seems strange given the existence of mind inspectors [7], advanced debugging techniques [8, 9], and a range of analytical tools [10, 11]. However, after delving deeper, it became apparent to us that the criticisms were directed more closely towards the quality of the Integrated Development Environments (IDEs) provided and their limitations in terms of practical features such as code completion, code navigation and formatting support. Over the years, it has become apparent that developers become uneasy when stripped of their traditional supports and that this engenders a feeling that the languages are not production quality.

Conversely, the perceived learning curve is less unexpected. AOP, with its origins in Distributed Artificial Intelligence, is underpinned by a quite different set of concepts to mainstream software engineering, where there is a clear evolution from procedural programming languages to Object-Oriented Programming (OOP) languages. Individuals attempting to learn about AOP are confronted with a range of concepts - beliefs, desires and intentions; speech acts; plans - that bear little relation to mainstream programming concepts. For many, this can act as a significant barrier to learning how to program in an AOP language.

Perhaps the most common explanation of the relationship between AOP and OOP is the comparison table presented in [12]. This table presents a very high-level view of AOP and OOP that treats AOP as a specialisation of OOP. Unfortunately, it provides little practical detail. For example, how does the state of an object relate to the state of an agent? is there any correlation between how behaviours are specified in OOP and how they are specified in agents? when and how will a behaviour be executed?

Answering these questions requires a more detailed comparison of AOP and OOP. However, when attempting to create a deeper comparison, it quickly becomes evident that it is not possible. The main reason for this is that AOP, unlike OOP, does not promote or enforce a consistent conceptual model (i.e. a standard view of state, methods, messages, etc.). Instead, different languages can, and are, based around quite different approaches. For example, AgentSpeak(L) style languages are essentially event-driven languages. They define context-sensitive event handlers that map events to partial plans. Conversely, GOAL is, at its heart, an action selection language where rules identify the context in which each action should be executed. The consequence of this diversity is that it is more appropriate to compare specific styles of AOP language with OOP rather than trying to over-generalise.

In this paper, we focus on understanding the relationship between AgentSpeak(L) and OOP with the goal of trying to reduce the perceived cognitive gap. Our approach is to start by identifying a mapping between AgentSpeak(L) and OOP concepts in Section 2, which we then reflect on in Section 3. The purpose of the reflection is to try to understand how to improve the design of AgentSpeak(L) to better support developers wishing to learn the language. Following on from this, we introduce a new member of the AgentSpeak(L) family called ASTRA that has been designed in response to the findings of the previous section. Full details of ASTRA are not provided in this paper. Instead, we focus on

only the most pertinent features. Finally, in Section 4 we present the relevant results of a wider survey carried out on the most recent class of the M.Sc. in Advanced Software Engineering.

2 Relating AgentSpeak(L) to OOP

AgentSpeak(L) can be prosaically described as an event-driven language where event handlers are fired based on both the triggering event and some context. Events, which are either external (environment-based) or internal (goal-based), are generated and added to an event queue. Events are then removed from this queue and matched to a rule which is then executed. The matching process checks both that the rule applies to the event and that the rule can be executed based on a rule context that defines valid program states in which the rule may be applied.

More commonly, the event handlers are known as **plan rules**; the program state is modeled as a set of **beliefs**, that are realized as atomic predicate logic formulae; the **events** are also modeled as atomic predicate formulae (with some additional modifiers); and the execution of plan rules is achieved through creation and manipulation of **intentions**. Finally, external events are generated through changes to the agent's state (i.e. the adoption or retraction of a belief), and internal events are generated through the declaration of **goals**.

It follows then, that an AgentSpeak(L) agent consists of an event queue, a set of beliefs (state), a set of plan rules (event handlers), and a set of intentions that are used to represent the execution of plan rules. Given that AOP is commonly viewed as a specialization of OOP, and that agents are a special type of object, the following outlines how AgentSpeak(L) concepts relate to OOP concepts from the perspective of an OOP developer:

Beliefs are equivalent to fields As indicated above, beliefs form the state of an agent. In OOP, state is defined in terms of a set of **fields** that hold values (or object references). If we consider a field, such as `int value;` this could be modeled as a belief `value(0)`. Here, the value 0 is chosen as it is the default value for integer fields in many OOP languages. To be fully precise, beliefs and fields are not the same. Whereas fields can be modeled using beliefs, beliefs actually encompass more than this, including environment information, global variables, etc.

Plan Rules are equivalent to methods A plan rule associates a plan with a triggering event and a context. Plans define behaviours and are basically blocks of procedural code that are executed whenever a matching event is processed and the rules context is satisfied. In OOP languages, procedural code is defined within methods and is executed whenever the method signature is matched to a message that has been received by the object. Accordingly, the AgentSpeak(L) equivalent of a method signature is the triggering event (specifically the identifier and the number of arguments). The context has no real equivalent in OOP, however, it can be viewed as providing

a form of method overloading based on state (i.e. when there are multiple rules matching a given event, the context is used to identify which of the rules should be executed).

Goals are equivalent to method calls Goals generate events. The are then matched to rules, which are subsequently executed. Method calls generate messages that are matched to methods that are executed. Typically, goals are declared from within a plan. The result is that the plan component of the selected rule is pushed onto the program (intention) stack and executed.

Events are equivalent to messages Within AgentSpeak(L), events play a similar role to messages in OOP. Events are used to trigger plan rules in the same way that, for OOP languages, messages are used to invoke methods. This can be somewhat confusing because "message" is also the term used for communication between agents, however this is not the focus here. In OOP, the set of messages that can be handled by an object is known as the **interface** of the object. This set of messages corresponds to the signatures of the methods that are defined in the objects implementing class(es). Given our view of events being equivalent to OOP messages, then in AgentSpeak(L) the interface of an agent is the set of events that it can handle.

Intentions are equivalent to threads Intentions represent the plans that the agent has selected based upon the matching of events to plan rules. The AgentSpeak(L) interpreter processes the intentions by executing the instructions contained within the plan. In cases where the instruction is a sub-goal, this results in an additional plan being added to the intention which must be executed before the next instruction in the initial plan can be executed. In most programming languages, this activity is modelled by the program (call) stack. Intentions are simply the AgentSpeak(L) equivalent of this. Given that an agent can have multiple concurrent intentions whose execution is interleaved, it is natural to view an intention as being the equivalent of a thread.

The above mappings are intended to relate the concepts of AgentSpeak(L) to those present in OOP. The objective behind this is to try to reduce the cognitive gap faced by individuals who know OOP and wish to learn an AOP language. The benefit of doing this is that someone who is proficient in OOP can use these mappings as a starting point for their study of the language.

3 Exploring the Implications

The mapping developed in Section 2 is not only potentially useful to developers aiming to learn AgentSpeak(L), but it is also useful from a language developer's perspective as it raises questions about the set of features that may be appropriate for AgentSpeak(L)-style languages. In this section, we explore some of the consequences of adopting the above mapping.

3.1 Beliefs are like fields

Understanding the role of beliefs in AOP languages can be one of the most challenging concepts to grasp. Certainly, at a high-level it is clear that beliefs

are the state, but many find it difficult to understand how beliefs relate to the state of an object. As was discussed above, one simple way of associating beliefs with object state is to demonstrate that beliefs are like fields. Fields are OOP's mechanism for defining the state of an object. Fields typically associate a label with a container for values, for example `String name = "Rem";` associates the field name, of type String with the value "Rem", which is itself a String literal. In AgentSpeak(L), it is possible to do something similar, namely to declare a fact, whose predicate corresponds to the field name, and which takes a single argument, the value associated with the field, for example `name("Rem");`.

In OOP, there are a couple of operations that can be performed on a field: (1) assigning a new value, for example, `name = "George";`; and (2) comparing a value, for example `name.equals("Rem")`. In AgentSpeak(L), performing these operations can be achieved as follows: (1) to assign a new value, you must first drop the existing belief and then adopt a new belief with the new value, for example, `-name("Rem");+name("George");` and (2) to compare the value, you can either perform a query of the agents beliefs, for example, `?name("Rem")` or as part of a plan rule context, for example `<te> : name("Rem") <- ...`. It should be noted here that the assignment operation, which is an atomic operation in OOP is not an atomic operation in AgentSpeak(L).

An interesting observation of the above is that, in transitioning from OOP (nominally Java) to AgentSpeak(L) the type of the field has been lost. Types can be a powerful feature of a programming language that can be used to statically verify the correctness of code. Specifically, in OOP, they can be used to identify situations where the wrong type of data is assigned to a field, or where the wrong type of data is passed to a method. Typically, AOP languages have used dynamically typed variables - this reflects the logical origins of AOP, where dynamically typed variables are common. For some developers, who come from a background where the languages they have used are strongly typed, this can be another significant hurdle to overcome.

One option for AOP language developers is to introduce a **type system** to their language [13]. Within AOP, type systems can be applied at the (multi-)agent level, and at the language level. (Multi-)agent types refer to the association of types with agent instances, which can be used for engendering reuse [14] of agent code or to support run-time substitution of agent instances [15].

The second use of type systems is to apply types to the terms of logical formulae, as is typically done in ontology languages, such as RDF [16]. The potential benefits of this are:

– **improved readability:** the meaning of the belief is clearer when the types are known.
– **static type checking:** compile-time checks can be used to reduce the number of run-time errors.

To take full advantage of static typing, a number of additional supports are required: correct forms for beliefs and (potentially) goals must be specified using an ontological representation; signatures representing the potential actions must be be specified in a similar way.

Pseudo code

```
1  Algorithm SelectionSort(A, n):
2    for j=1 to n-1 do
3      minIndex = j
4      for k=j+1 to n-1 do
5        if (A[minIndex] < A[k]) then
6          minIndex =k
7      if (minIndex <> j) then
8        temp = A[j]
9        A[j] = A[j+1]
10       A[j+1] = temp
11   return A
```

AgentSpeak(L) code

```
1  !do_sort([7, 5, 12, 15, 3]);
2
3  +!do_sort(L) <-
4    _size(L, S);
5    !outerLoop(L, S, 0);
6    ?sorted(L2);
7    _print(L2).
8
9  +!outerLoop(L, S, X) <-
10   +min_index(X);
11   !innerLoop(L, S, X);
12   ?min_index(Z);
13   -min_index(Z);
14   !update(L, S, X, Z).
15
16 +!update(L, S, X, Z) : X < Z <-
17   _swap(L, X, Z, L2);
18   !outerLoop(L2, S, X+1).
19
20 +!update(L, S, X, Z) <-
21   !outerLoop(L, S, X+1).
22
23 +!outerLoop(L, S, X) <-
24   +sorted(L).
25
26 +!innerLoop(L, S, X) : X < S <-
27   _elementAt(L, X, T);
28   !compare(L, X, T);
29   !innerLoop(L, S, X+1).
30
31 +!innerLoop(L, S, X) <-
32   _skip().
33
34 +!compare(L,X,T):min_index(Y)<-
35   _elementAt(L, Y, S);
36   !compare(L, X, Y, S, T).
37
38 +!compare(L, X, Y, S, T)
39     : S < T <-
40   -min_index(Y);
41   +min_index(X).
42
43 +!compare(L, X, Y, S, T) <-
44   _skip().
```

Fig. 1. Two implementations of Selection Sort algorithm

3.2 Plans Rules as Methods

The equivalence of plan rules and methods posits a simple question: 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 agent language?

To investigate this in more detail, we decided to implement a common algorithm using AgentSpeak(L). The choice of algorithm itself is not important, as the question really being asked here is: can somebody learning an AOP language apply their existing algorithmic problem solving skills easily in that language?

The result is illustrated in Figure 1. The left hand piece of code is standard pseudo code for the selection sort algorithm. The right-hand piece of code is the AgentSpeak(L) implementation of that algorithm. As can be seen, the AgentSpeak(L) solution is far more complicated than the pseudo code - it is over 3 times longer; one method has been mapped to 9 rules (the first rule in the AgentSpeak(L) program actually calls the sorting algorithm); and it is not even all of the code because 5 primitive actions are used (`_size(...)`, `_elementAt(...)`, `_swap(...)`, `_print(...)`, and `_skip()`). In fact, there are a number of clear issues with the AgentSpeak(L) solution:

1. **Rule explosion** occurs because in AgentSpeak(L) loops and selections are implemented using rules. In fact, 2 rules are typically required for both if statements and loops. In both cases, one rule is required where the guard is true and one where the guard is false. Both rules must be provided in all cases, even if they do nothing (failure to match an internal event to a rule is equated to failure to achieve a sub-goal as there are no valid event handlers for the given event).
2. **Returning results** is an issue in AgentSpeak(L) because the basic version of the language does not allow values to be returned from a sub-goal call. Instead, the value must be stored in a belief (in the global state) and upon completion of the sub-goal, the value must be retrieved by querying the global state. Such a convoluted approach clearly is not scalable given AgentSpeak(L) supports multiple concurrent intentions.
3. **Hidden code** arises because AgentSpeak(L) has such limited semantics that it is not able to directly perform simple operations such as swapping two values. Instead a number of custom primitive actions are also needed (these are not included in the code count) to implement this basic functionality. In the code any statement that is prefixed by a _ is a primitive action.
4. **Loss of readability** due to the number of rules and the convoluted control flow that results from it understanding the agent code is far more difficult than understanding the pseudo code.

Admittedly, many would question the value in implementing a sorting algorithm using an agent language, but again, the issue here is not the actual algorithm, but that algorithms cannot be easily implemented in AgentSpeak(L). Given the amount of time and effort that is put into teaching programmers to think algorithmically, it seems inefficient to be promoting languages that do not try to leverage those skills.

3.3 Intentions as Threads

In the mapping, we equate intentions with threads. Agents are commonly presented as being active objects, with their own thread of control. However, the reality is that AgentSpeak(L) agents are more like multi-threaded processes, with each intention being an individual thread. If this view is adopted as the correct analogy for intentions, then our languages must be designed with this in mind.

AgentSpeak(L) is not designed with such a view in mind. As was mentioned above, sub-goals cannot return values. Instead, the value must be stored in the global state of the agent and retrieved once the sub-goal has completed. It is easy to see that such a scenario does not work well if intentions are like threads. This is especially the case since intentions are normally interleaved, with the agent executing one action for one intention per iteration of its execution cycle.

Consider, for example, an agent with two intentions, A and B, that both need to sort a (different) list of numbers using the selection sort code of figure 1. On iteration i, intention A stores the sorted list in its global state. On the next iteration (i+1), intention B stores its sorted list in the global state. Two iterations later (after A and B have completed their sub-goals), A then attempts to retrieve the sorted list from the memory. The agent has two beliefs - one for each sorted list - based on the given program, it is ambiguous as to which of the sorted lists will be returned. The result is that either A or B will have the incorrect sorted list. Naturally, this problem can be overcome, but only by further increasing the complexity of the program!

One solution would be to introduce support for mutual exclusion into AgentSpeak(L). This would overcome the issue, but would require the mutual exclusion to be applied prior to the first invocation of the `!outerLoop(L, S, X)` sub-goal. The natural alternative is to allow sub-goals to return values.

3.4 Events are like messages

Perhaps the most contentious part of the mapping is the association of AOP events and OOP messages. This can seem contentious because messages are a well-defined concept in multi-agent systems that drive speech act based interaction between agents. Further, it conflicts with Shoham's analysis, which argues that message passing in AOP is equivalent to message passing in OOP. In reality, there is no conflict. The reason for the seeming inconsistency is that Shoham compares agents and objects from an external (and high-level) perspective, whereas our comparison of AgentSpeak(L) and OOP is more low-level. Further, the design of AgentSpeak(L) did not consider inter-agent communication.

There are two basic approaches to handling the receipt of messages in AgentSpeak(L). The first approach is the approach adopted in Jason. Here, a subset of KQML is identified and the chosen speech acts are closely integrated with the language. For example, receipt of a **tell** message results in the adoption a belief based on the content of the message together with an annotation identifying the sender of the message. Invoking a behaviour based on the receipt of a tell message thus requires the creation of a plan rule whose triggering event matches the belief adoption event created by the receipt of the message. The sending of messages is then supported through the provision of an internal action `.send(...)`. This approach fits the mapping presented in this paper because the semantics of the receipt of messages are hidden from the programmer.

An alternative approach is to introduce a new **message event** type to model the receipt of a message. This approach is more loosely coupled in that the receipt of a message does not have a direct impact on the agent. Instead, the programmer

must implement a rule to handle the receipt of the message. The advantage of this approach is that it is left to the programmer to determine how the agent responds to the receipt of a message. For example, if an agent is informed of some new fact, then the programmer can provide a rule to define whether or not the agent should adopt the content as a belief. As before, sending of messages can be achieved through a custom action (or alternatively, a custom plan operator).

Irrespective of the model chosen, it is clear that AOP messages are not the same as OOP messages as ultimately, the behaviour resulting from the receipt of the message is realised through the processing of an event. What is interesting to note from the second model is the idea of increasing the number of event types supported by the language. The benefit of adding new event types is that the events can be specified in a way that all of the relevant data is encoded in the event. This can result in a solution that is clearer and easier to follow that trying to reduce every event to an annotated belief. The cost comes from the fact that the implemented language must handle more event types.

4 ASTRA: AgentSpeak(L) with bells and whistles

The mapping presented in this paper is aimed at reducing the cognitive gap for developers who are familiar with OOP and who wish to learn an AOP language. In order to evaluate whether such a mapping can help, we have developed a new implementation of AgentSpeak(L) called ASTRA. ASTRA is based upon Jason, but includes a number of features that are inspired by the mapping presented in this paper. In line with the rest of this paper, the syntax of ASTRA is based upon Java syntax, which has been chosen so that the language will seem more familiar to the user. In this section, we present only the most pertinent details of ASTRA that reflect the points made in the paper. For more information on the language, the reader is directed to [6].

4.1 The ASTRA Type System

ASTRA as a statically typed language that provides a typical set of primitive types for use. Because ASTRA is built on Java, and in an effort to improve the cohesion between the agent layer and the supporting functionality in the Java layer, the set of primitive types is based upon Java's type system. While not exhaustive, all the necessary types are provided for, including 4 and 8 byte integers (mapped to Java's `int` and `long` types), 4 and 8 byte floating point numbers (mapped to `float` and `double` types) as well as representations for character and boolean values (mapped to `char` and `boolean` types).

In addition, ASTRA also supports the non-primitive types of character strings which maps to the Java `String` class and a list type which maps to a custom implementation of the `java.util.List` interface. Finally, ASTRA allows the use of generic objects through the object type. Instances of objects cannot be directly represented within the language but can be stored and passed to internal and environment operations.

ASTRA uses modules to represent internal libraries. The design of these libraries is inspired by the use of annotations in CArtAgo [17]. Libraries allow four kinds of annotation: terms, formulae, sensors and actions. Terms represent basic calculations that can return a value. Formula methods are constructors that return any logical formula instance in ASTRA (these can be simple boolean values or more complex formulae). Sensors generate beliefs that are added to the agent's state. Actions represent internal actions that can be performed, returning a boolean value indicating if the action was successfully performed. Figure 2 shows the declaration of a module containing a single term and action.

All of the components of the modules are typed. This enables the static verification of types for any usage of the library as well as for any value returned. Terms, actions and formulae can be used in a manner intuitive to OOP programmers: Figure 3 shows an example of the use of a term to determine the largest of two numbers before using an action to print it.

Modules must first be declared by linking the class to a name within the agent, this declaration is shown in line 5 of the example. A consequence of this method of declaration is that a single agent can create several copies of the same module, each with a different name and state.

```
1  package ex;
2
3  import astra.core.Module;
4
5  public class MyModule extends
       Module {
6
7   @TERM
8   public int max(int a, int b){
9    return Math.max(a, b);
10  }
11
12  @ACTION
13  public boolean printN(int n){
14   System.out.println(n);
15   return true;
16  }
17 }
```

Fig. 2. Java code declaring a module with a term and action

```
1  package ex;
2
3
4  agent Bigger {
5    module MyModule m;
6
7
8    initial num(45, 67);
9    initial !init();
10
11
12   rule +!init() {
13     query(num(int X,int Y));
14     int n = m.max(X,Y);
15     m.printN(n);
16   }
17
18 }
```

Fig. 3. ASTRA code declaring and using a module

It should be noted that ASTRA is not alone in considering strong typing to be important in agent programming. The simpAL agent programming language [13] also supports typing, and includes the ability to extend strong typing to environment artifacts and to the agents themselves.

4.2 Extended Plan Syntax

ASTRA includes a number of extensions to the traditional AgentSpeak(L) plan syntax. These extensions are added to combat the issues noted in Section 3.2.

The usefulness of constructs such as these is emphasised by Jason's inclusion of some of these procedural-style constructs (e.g. if statements, loops) in its extended version of AgentSpeak(L). ASTRA attempts to provide a more complete mapping between procedural-style pseudocode, as well as AOP features.

If statement the most basic form of flow control
While loop traditional method of repetition in programming
Foreach loop repeats the same actions for every matching binding of a formula
Try-recover block 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 bind 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 provides for mutual exclusion in critical sections

Figure 4 shows an implementation of selection sort as a single rule in ASTRA. While this demonstrates only some elements of the extended plan syntax, when compared to the Agentspeak(L) implementation given in Figure 1 it is much easier to understand.

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

Fig. 4. ASTRA rule for Selection Sort

4.3 Mutual Exclusion Support

In Section 3.3, the link between intentions and threads was established. This introduces potential difficulties in the form of race conditions since multiple intentions are, interleaved by their very nature. As such, it is necessary to provide functionality to offset these difficulties. To facilitate removal of these high-level

race conditions, ASTRA includes support for synchronized blocks - sections of the agent program that are labeled as critical sections.

Code contained within a synchonized block can only be executed by a single intention at a time. Synchronized blocks are declared using the synchronized keyword but also require an identifier for the block. This allows multiple blocks to be declared representing a common critical section. Once an intention enters a synchronized block, all synchronized blocks with the same identifier are locked and cannot be entered until the current intention is completed.

Figure 5 shows an example of ASTRA code with race conditions. This program invokes the `!init()` goal twice, creating 2 intentions. In this situation, there is no way to know the output of the program. If both intentions query the belief at the same time the agent will only output the value of X at 0 and 1 (initial and incremented once). Figure 6 shows the same program with mutual exclusion added through the use of a synchronized block. In this situation, the output is guaranteed show the values of X at 0, 1 and 2.

```
1  agent Racy {
2    module Console C;
3
4    initial ct(0);
5    initial !init(), !init();
6
7
8    rule +!init() {
9      query(ct(int X));
10     +ct(X+1);
11     -ct(X);
12   }
13
14
15   rule +ct(int X) {
16     C.println("X = " + X);
17   }
18 }
```

Fig. 5. ASTRA code with race conditions

```
1  agent Racy {
2    module Console C;
3
4    initial ct(0);
5    initial !init(), !init();
6
7    rule +!init() {
8      synchronized (ct_tok) {
9        query(ct(int X));
10       +ct(X+1);
11       -ct(X);
12     }
13   }
14
15   rule +ct(int X) {
16     C.println("X = " + X);
17   }
18 }
```

Fig. 6. ASTRA code with mutual exclusion

5 Evaluation

In order to evaluate the concepts discussed in this paper, a survey was conducted using 20 students from the M.Sc. in Advanced Software Engineering programme in University College Dublin. The participants completed an Agent-Oriented Software Engineering module as a component of their degree. The students are full-time software engineers with an average experience of 7.65 years in industry. The degree is completed part-time over a number of years where each module is taught intensely for a single week of lectures and practical instruction.

Participants were asked to indicate their level of agreement with statements relating to general agent-oriented programming as well as more specific areas of interest to this work. The results were captured on a 5-point Likert scale. The questions presented form only a subset of the overall survey that was performed, full details of the survey and responses can be view in [18, pp. 181–187].

The first group of questions relate to agent-oriented programming languages in general and their benefit to the participant.

Q1 Agents are a useful level of abstraction.
Q2 I would consider using an AOP language in my (future) work.
Q3 AOP languages make distributed programming easier.
Q4 AOP languages make concurrent programming easier.
Q5 Studying AOP languages enhanced my understanding of distributed computing

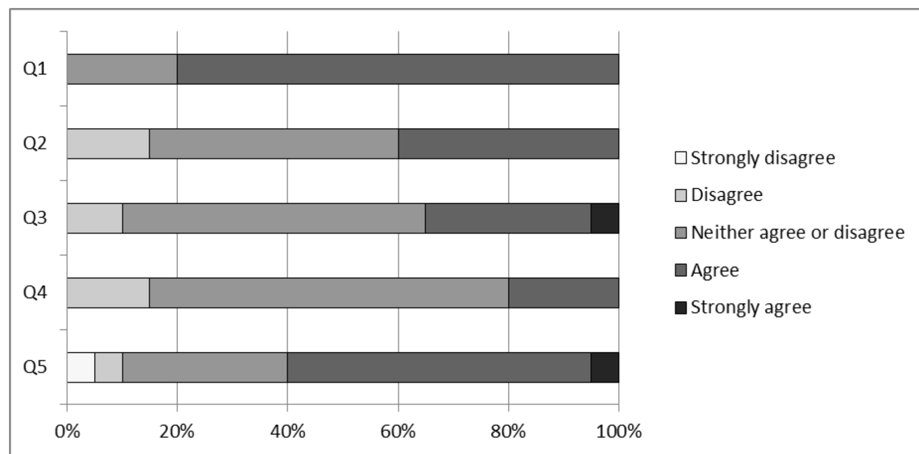

Fig. 7. Representation of Agents Survey results

The second group of questions relate to the participants' perception of some features of ASTRA and the impact this had on their ability to learn the language.

Q6 Static typing, and the verification this enables, are important.
Q7 Static typing is a necessary feature of AOP languages.
Q8 Static typing makes ASTRA code easy to read.
Q9 ASTRA was easy to learn.
Q10 I found it easy to apply my existing programming knowledge to ASTRA.
Q11 The syntax of ASTRA made it easy to understand.
Q12 There is a steep learning curve for ASTRA.
Q13 The lack of a debugger made ASTRA more difficult to learn.
Q14 ASTRA offers a good level of abstraction for programming distributed systems.

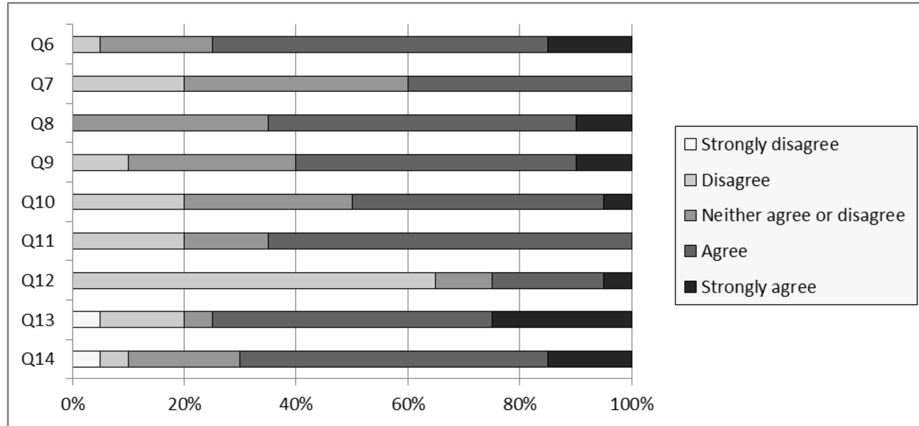

Fig. 8. Representation of ASTRA Survey results

The results from the first group of questions show that the participants generally consider agents to be a useful level of abstraction. However, only 40% indicated that they would consider using AOP in their future work. This may indicate that further work is required to address the criticisms raised Section 1.

The results from the second group of questions show that 75% of the participants believe that static typing is important. However, of this 75%, only 53% believe that it is a necessary feature for AOP languages. Regardless of the necessity of static typing, 65% of the participants believed that static typing made ASTRA code easier to read. Further, 65% of the participants also found the syntax of ASTRA made it easy to understand. This may indicate that provision of constructs more commonly available in OOP languages eases the transition from procedural languages to ASTRA.

In terms of learning ASTRA, the majority of participants disagreed that there is a steep learning curve. However, 75% of participants did believe that the lack of a debugger made the learning process more difficult. While this agrees with the informal feedback received previously, it is unclear whether this is a result of the level of experience of the participants. The amount of industry experience may make it more likely that the participants would utilise this level of support more than less experienced students of AOP languages.

6 Conclusions

In this paper, we have presented a practical conceptual mapping between AgentSpeak(L) and Object-Oriented Programming (OOP). The purpose of this mapping has been to attempt to find a way of reducing the cognitive gap for developers, experienced in OOP, who wish to learn Agent-Oriented Programming (AOP). In developing the mapping, we are not attempting to reduce one

paradigm to the other, but instead aim to provide a stepping stone that will help developers wishing to learn AOP make their first steps.

In addition to the benefit such a mapping provides for those wishing to learn AgentSpeak(L), a second benefit is that it provides language designers with valuable insights into how their languages might be used in practice. To this end, Section 3 reflects on the mappings and identifies a number of possible issues and potential opportunities:

1. the potential of using a type system to improve the link between the agent and object layers and to reduce run-time defects through static type checks.
2. the provision of an extended suite of plan operators including a subset that mirror the typical constructs offered in procedural languages to support the use of existing algorithmic problem solving skills when developing agent behaviours and the curtailing of rule explosion that was evident in Figure 1.
3. the provision of mutual exclusion support for intentions to facilitate management of critical sections.
4. the use of an extended suite of event types rather than attempting to force all events to conform to AgentSpeak(L)'s original model of belief and goal events.

While we believe that we have come to these conclusions through a novel route, we do not claim to be the first to reach them. Certainly, Jason includes support for atomic behaviours and has an extended suite of plan operators. In terms of the latter, we do believe that our perspective offers some benefit: while Jason does include support for `if` statements and `for` and `while` loops, we do not believe that it offers support for local variable declaration or assiginment, both of which are considered a core concept in pseudo code.

The principal outcome of our work has been to drive the development of ASTRA, an implementation of AgentSpeak(L) that is targeted towards reducing the cognitive gap. In 2014, ASTRA was made available to students on the M.Sc. in Advanced Software Engineering mentioned in the introduction. Students learned ASTRA over a 5-day period, during which they wrote a range of programs. On the last day, they were assigned a complex problem to solve [18, pp. 167–180] and were asked to complete a questionnaire relating to both the problem and more generally agents. Details of the results of the relevant parts of this questionnaire are presented in Section 5. We believe that the feedback positively reflects our decision to include both the language level type system and the suite of plan operators into ASTRA.

References

1. Dastani, M.: 2APL: a practical agent programming language. Autonomous Agents and Multi-Agent Systems **16**(3) (2008) 214–248
2. Dastani, M., van Birna Riemsdijk, M., Meyer, J.J.C.: Programming multi-agent systems in 3APL. In: Multi-agent programming. Springer (2005) 39–67

3. Hindriks, K.V.: Programming Rational Agents in GOAL. In El Fallah Seghrouchni, A., Dix, J., Dastani, M., Bordini, R.H., eds.: Multi-Agent Programming:. Springer US (2009) 119–157
4. Bordini, R.H., Hübner, J.F., Wooldridge, M.: Programming multi-agent systems in AgentSpeak using Jason. John Wiley & Sons (2007)
5. Russell, S., Jordan, H.R., O'Hare, G., Collier, R.W.: Agent Factory: A Framework for Prototyping Logic-Based AOP Languages. In Klügl, F., Ossowski, S., eds.: Multiagent System Technologies. Volume 6973 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2011) 125–136
6. : Astra language website. `http://www.astralanguage.com/` Accessed: 2015-06-21.
7. Collier, R.W.: Debugging agents in agent factory. In: Programming Multi-Agent Systems. Springer Berlin Heidelberg (2007) 229–248
8. Hindriks, K.V.: Debugging is explaining. In Rahwan, I., Wobcke, W., Sen, S., Sugawara, T., eds.: PRIMA 2012: Principles and Practice of Multi-Agent Systems. Volume 7455 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2012) 31–45
9. Lam, D.N., Barber, K.S.: Debugging agent behavior in an implemented agent system. In Bordini, R.H., Dastani, M., Dix, J., El Fallah Seghrouchni, A., eds.: Programming Multi-Agent Systems. Volume 3346 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2005) 104–125
10. Botia, J.: Debugging huge multi-agent systems: group and social perspectives. (2005)
11. Doan Van Bien, D., Lillis, D., Collier, R.W.: Space-time diagram generation for profiling multi agent systems. In: Programming Multi-Agent Systems. Springer Berlin Heidelberg (2010) 170–184
12. Shoham, Y.: Agent-oriented programming. Artificial intelligence **60**(1) (1993) 51–92
13. Ricci, A., Santi, A.: Typing multi-agent programs in simpal. In Dastani, M., Hübner, J.F., Logan, B., eds.: Programming Multi-Agent Systems. Volume 7837 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2013) 138–157
14. Dhaon, A., Collier, R.W.: Multiple Inheritance in AgentSpeak (L)-Style Programming Languages. In: Proceedings of the 4th International Workshop on Programming based on Actors Agents & Decentralized Control, ACM (2014) 109–120
15. Baldoni, M., Baroglio, C., Capuzzimati, F.: Typing multi-agent systems via commitments. In Dalpiaz, F., Dix, J., van Riemsdijk, M.B., eds.: Engineering Multi-Agent Systems. Volume 8758 of Lecture Notes in Computer Science. Springer International Publishing (2014) 388–405
16. Emmons, I., Collier, S., Garlapati, M., Dean, M.: Rdf literal data types in practice. In: The 7th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS 2011). 1
17. Ricci, A., Viroli, M., Omicini, A.: CArtAgO: A Framework for Prototyping Artifact-Based Environments in MAS. In Weyns, D., Parunak, H.V.D., Michel, F., eds.: Environments for Multi-Agent Systems III. Volume 4389 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2007) 67–86
18. Russell, S.: Real-time monitoring and validation of waste transportation using intelligent agents and pattern recognition. Phd thesis, University College Dublin (2015)