Exploring AOP from an OOP Perspective

Rem W. Collier
School of Computer Science
University College Dublin
Belfield, Dublin 4, Ireland
rem.collier@ucd.ie

Seán Russell
School of Computer Science
University College Dublin
Belfield, Dublin 4, Ireland
sean.russell@ucd.ie

David Lillis
School of Computer Science
University College Dublin
Belfield, Dublin 4, Ireland
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 between Object-Oriented Programming (OOP) and the AgentSpeak(L) family of AOP languages. This mapping explores how OOP concepts and the concurrent programming concept of threads relate to AgentSpeak(L) concepts. We then use our analysis of this mapping to drive the design of a new programming language entitled ASTRA.

Categories and Subject Descriptors

I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence—*Multiagent systems, Languages and structures*; D.3.2 [**Programming Languages**]: Language Classifications—*Multiparadigm languages*

General Terms

Languages, Theory

Keywords

Agent-Oriented Programming, AgentSpeak(L), ASTRA

1. INTRODUCTION

The Agent-Oriented Programming (AOP) paradigm is now almost 25 years old. Since its inception, a number of established AOP languages have emerged, with the most prominent being: 2/3APL [8, 9], GOAL [12] 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 [22], Jason [4], and our most recent agent-programming language, ASTRA [1].

Each year, the students have provided 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 [6], advanced debugging techniques [13, 16], and a range of analytical tools [5, 11]. However, after delving deeper, it became apparent that the criticisms were directed more 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 [23]. 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,

AGERE 2015 Pittsburgh, PA, USA

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. We begin by identifying a mapping between AgentSpeak(L) and OOP concepts in Section 2, which we 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. In response to our analysis, Section 4 introduces a new member of the AgentSpeak(L) family called ASTRA. Full details of ASTRA are not provided in this paper. Instead, we focus on only the most pertinent features.

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 by declaring **goals**.

It follows 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 represent the execution of plan rules. Given that AOP is commonly viewed as a specialisation of OOP, and that agents are a special type of object, it is possible to relate AgentSpeak(L) concepts 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 Events are generated due to adoption or retraction of goals. These are then matched to rules, which are subsequently executed. Method calls generate messages, which 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 The events that are part of AgentSpeak(L) 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 as 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");`. This process has been optimised in Jason to `+-name("George")`, where `name("George")` is adopted and all previous predicates matching `name(X)` are removed. This optimisation is very useful in a situation where a single belief is being used in the same way as a global variable might be. In order to (2) 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 and Jason, 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. Within AOP, it is possible to apply type systems at two levels: the (multi-)agent level, and at the language level. (Multi-)agent types refer to the association of types with agent instances, such types can be used for engendering reuse [10] of agent code or to support run-time substitution of agent instances [3].

The second use of type systems is to apply types to the terms of logical formulae. 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 could be specified using an implementation specific mechanism; for example a list of valid predicate formula signatures. For example, these could be specified in a manner similar to an `actionspec` in GOAL which is used to specify pre- and post-conditions for actions [14].

This requirement can be extended further to encompass environment interaction. In this case, the use of an environment interface such as CArtAgO [20] would also require that types be specified. In simpAL this takes the form of an `artifactmodel` which describes the usage interface of all artifacts implementing that model [18]. This allows static type checking in any use of an artifact. However, it does not does not extend to any events that may be generated by the artifact.

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

```
Algorithm SelectionSort(A, n):
  for j = 1 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
```

Pseudo code

```
!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().
```

AgentSpeak(L) code

Figure 1: Two implementations of Selection Sort algorithm

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

```
!init(1).
!init(2).

+!init(X) <-
    print(X+"A");
    print(X+"B");
    print(X+"C).
```

Figure 2: AgentSpeak(L) Interleaving Example

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 [15]. The reality is that implementations of an agent can vary from a single threaded architecture to highly complex multi-threaded architecture. Here, we do not focus on such low level issues, instead, we explore AgentSpeak(L) at a higher level.

In AgentSpeak(L), the execution of behaviours is modeled through intentions. An agent creates a new intention for every external event that it matches to a plan rule (it also generates a new intention for each initial goal that is declared in the program, but this is a special case for goal events). An intention is basically an execution stack - it contains each action that must be performed in order to achieve the intention, with the next action to be performed sitting on top of the stack. An execution step involves removing the top item from the stack and executing it. In situations where an agent has multiple intentions, the accepted view is that intention execution is interleaved - on each iteration, one intention is selected and executed.

Consider the sample program in Figure 2. On creation, two goal events are generated `+!init(1)` and `+!init(2)`. On the first iteration of the agent interpreter, the first event is handled, resulting in the adoption of an intention that contains three print actions. Given that this is the only intention of the agent, the first step of this intention is also executed on the first iteration. On the second iteration, the second event is handled, resulting in a second intention that also contains three print actions. At this point, there is some uncertainty as to what happens as it is not clear which intention would be selected for execution. To remove ambiguity, let us assume that new intentions should always be selected for execution. The result is that the first action of the second intention is now executed. On subsequent iterations, a round robin execution policy can be enforced, allowing fair use of the underlying processors. This means that the first intention will be scheduled on step 3 and the second intention on step 4. This interleaving will continue while the agent has multiple intentions. The resulting output is "1A 2A 1B 2B 1C 2C". Note that, even if we had adopted a policy where new intentions are not executed on the step they are created, then the output would have been "1A 1B 2A 1C 2B 2C" - so interleaving still occurs.

From this above example, it is clear that, irrespective of the actual threading model used, an AgentSpeak(L) agent with multiple intentions is like a process that has multiple threads where each intention is akin to a 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 particularly given their execution can be interleaved.

This can be easily illustrated by considering 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 approach to handling interleaved execution of intentions is 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 causing the second intention to be delayed until the first has completed. The simpler option is to allow sub-goals to return values.

Jason includes support for mutual exclusion through the `atomic` keyword. This keyword can be applied to plan rules and causes the rule to be treated like a synchronized method. Once an atomic rule is executed, all other atomic rules are blocked until the active atomic rule has completed. While this functionality is sufficient to address mutual exclusion, we are not convinced that it provides the flexibility necessary for more complex applications. Our concerns are specifically focused on two issues:

- *Lack of support for multiple critical sections.* Mutual exclusion ensures sequential execution of code where required. This can result from the need to control access to a resource or simply to ensure that some sequence of actions is performed atomically. It is not unreasonable to assume that some agents will have more than one critical section. In such cases, enforcing agent-level mutual exclusion will result in intentions being blocked even though they are not accessing the same critical section. In languages like Java, such issues can be alleviated through the use of synchronized blocks.
- *Limited granularity of locking mechanism.* By providing only a rule level of mutual exclusion, developers

```
1 +event : context
2   <- ....
3      while(vl(X) & X > 10) { // where vl(X) is a
       belief
4        .print("value > 10");
5        -+vl(X+1);
6      }
7      ....
```

Figure 3: Sketch of While-loop in Jason

are required to implement critical areas as atomic plan rules. As was discussed in Section 3.2, this can lead to rule explosion and reduced readability of code.

In our view, a more appropriate level of support for critical sections is to allow synchronized blocks that have an associated token. This token would act like an identifier for a critical section, and the mutual exclusion mechanism would only block intentions trying to access a critical area whose token has been taken by another intention. These intentions would be blocked until the active intention releases the token by reaching the end of the synchronized block.

A third issue arising from the adoption of the view of intentions as threads is the use of global state to maintain local state. This is particularly an issue if extended plan operators are introduced as is recommended in Section 3.2.

The best way to illustrate this is to explore the example in Figure 3 which is taken from [2]. In this example, the code uses global state to represent a loop counter (`vt(X)` in the example). This is in line with our discussion in section 3.1, as is the increment on line 5, which uses the Jason belief update optimization. The problem with this is that global state is being used to implement something that would traditionally be implemented using a local variable. Furthermore, given the above discussion, the code clearly cannot be executed concurrently as the belief representing the loop counter is a critical section. This has the implication that, in Jason *every loop is (part of) a critical section.* This would have significant performance consequences if the mutual exclusion mechanism provided does not allow multiple critical sections to be defined. The natural consequence of the introduction of additional constructs, such as loops, is that AgentSpeak(L)-style languages require some mechanism to define local state as well as global state.

3.4 Events as 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, typically using some Agent Communication Language (ACL). Furthermore, 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.

Since our analysis associates AOP events with OOP messages, it is interesting to also compare how these events can come about. In OOP. an object's *interface* is typically described as the set of methods that can be invoked when messages are sent from other, external, objects. These methods are often described as being "public". The object itself may invoke any of these public methods, but also other methods that have been marked "private" (we will not consider "protected" methods, as inheritance is not common in AOP languages, though it does exist [10]).

In AgentSpeak(L), two types of events can be raised: belief events occur whenever beliefs are adopted or retracted, and goal events operate in a similar way for goals. The only entity capable of affecting the goals of an agent is itself. This means that a clear parallel can be drawn between plan rules that react to goal events and private methods in OOP.

The situation with belief events is somewhat more complex. Unlike goal events, beliefs can be created or removed due to external factors, in addition to the agent's own operations. For example, changes in the environment in which an agent is situated typically result in changes to the agent's belief base, resulting in the generation of belief events. In a similar way, the receipt of ACL messages is frequently implemented by updates to the agent's mental state, most commonly the adoption of beliefs. Because AgentSpeak(L) does not distinguish between "private" or "public" plan rules, and treats all belief events equally, it is difficult to draw a direct parallel between the public interface of an object and a similar concept for agents.

There are two basic approaches to handling the receipt of messages in implementations of 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 as 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 plan operator).

It is an open question as to which of these approaches is preferable, and a similar discussion could be had on how to cater for information arising from the agent's environment. One attractive element of the latter approach is that belief events would no longer be directly triggered by external elements such as the environment or other agents. The behaviour resulting from such an external event is realised through the processing of an event by the agent itself, and the beliefs it adopts (if any) in response. 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) ENHANCED

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 [1].

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).

ASTRA also supports the non-primitive types: character strings, which map to the `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 [20]. 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 4 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 5 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 }
```

Figure 4: 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 }
```

Figure 5: 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 [19] 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

```
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++;
  }
}
```

Figure 6: ASTRA rule for Selection Sort

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 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 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 enables mutual exclusion in critical sections

Figure 6 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.

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 synchronized 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.

```
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);
  }
}
```

Figure 7: ASTRA code with race conditions

```
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);
  }
}
```

Figure 8: ASTRA code with mutual exclusion

Figure 7 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 8 shows the same program with mutual exclusion added through the use of a synchronized block. In this situation, the output is guaranteed to show the values of X at 0, 1 and 2.

4.4 Extended Event Types

AgentSpeak(L) defines two basic types of event: belief update events, and goal events. This reflects the focus of the design of AgentSpeak(L) on the internal reasoning mechanism of an agent. Jason extends AgentSpeak(L) in numerous ways, one of which is the integration of support for agent interaction. The manner in which Jason performs this integration has been to define a limited set of message types and define the semantics of how the receipt of these messages

```
agent Test {
  module Console C;
  module System S;

  rule +!main(list args) {
    send(inform, S.name(), count(0));
  }

  rule @message(inform, string from, count(int X)) :
     X < 10 {
    C.println("X=" + X);
    send(request, S.name(), count(X+1));
  }

  rule @message(inform, string from, count(int X)) {
    C.println("X=" + X);
    C.println("STOPPED");
    S.exit();
  }
}
```

Figure 9: ASTRA agent that talks to itself

affects the state of the agent. For example, an `achieve` message causes a goal to be adopted by the receiving agent, and a `tell` message causes a belief to be adopted by the receiving agent. In order to capture additional information about the source of the belief / goal, Jason has introduced an annotation mechanism. For example, the sender of a `tell` is captured as a `source(X)` annotation that is appended to the generated belief. The advantage of this approach is that it allows additional functionality to be added to AgentSpeak(L) without the need to modify the set of event models. The disadvantage is that any additional information is stored as annotations. This, in effect, pushes the complexity of knowledge representation into the annotation mechanism.

An alternative approach is to introduce additional event types, allowing each event type to appropriately capture the information that is relevant to the event. For example, receipt of a message can be modeled as a separate event type, known as a *message event*. An example of this is illustrated in Figure 9, which illustrates a simple ASTRA agent that sends a message to itself. It is useful to note that, even though the agent receives an `inform` message (the ASTRA equivalent of a Jason `tell` message), the agent does not need to adopt a belief. Instead, it is left to the developer to implement code that does this. One arguable disadvantage is that an agent does not have any guarantees on the effect of informing another agent of something.

Overall, the use of extended event types can be, conceptually, less appealing than the Jason approach as it introduces another event type, but practically, we believe that this may be a better approach because it more clearly identifies plan rules that are intended to handle interaction with other agents. It also maintains a cleaner separation between beliefs - which come from internal actions or the environment - and messages. Finally, the approach can be used to further augment the functionality of the language and to maintain a clean separation of concerns between the core functionality and the new extended functionality. Such an approach has been used to provide integrated support for environments like CArtAgO [20] and extended conversation management functionality, such as ACRE [17].

5. 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 assignment, both of which are considered a core concept in pseudo code. Indeed, as was discussed in Section 3.3, lack of such support can in fact lead to an explosion of critical sections, which results in a widespread necessity for mutual exclusion.

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 5 days, during which they wrote a range of programs. What was interesting to note, from informal observation, was the size and complexity of programs written on the first day of the course. Previous years have seen program sizes that were typically less than 30 lines of code, whereas through the use of ASTRA program sizes typically increased dramatically to over 100 lines of code. The complexity of problems attempted was also much higher. It was also clear that students found it easier

to understand control flow within their programs, as their solutions were more procedural in nature at first. This relatively gentle introduction to agent programming meant that the students were in a better position to appreciate the benefits of the higher-level agent abstraction later in the week.

On the last day, the students were assigned a complex problem to solve [21, 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 [7]. 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.

6. REFERENCES

[1] Astra language website. `http://www.astralanguage.com/`. Accessed: 2015-06-21.

[2] Documentation for Jason while-loop implementation. `http://jason.sourceforge.net/api/jason/stdlib/loop.html`. Accessed: 2015-08-13.

[3] Matteo Baldoni, Cristina Baroglio, and Federico Capuzzimati. Typing multi-agent systems via commitments. In Fabiano Dalpiaz, Jürgen Dix, and M. Birna van Riemsdijk, editors, *Engineering Multi-Agent Systems*, volume 8758 of *Lecture Notes in Computer Science*, pages 388–405. Springer International Publishing, 2014.

[4] Rafael H. Bordini, Jomi F. Hübner, and Michael Wooldridge. *Programming multi-agent systems in AgentSpeak using Jason*. John Wiley & Sons, 2007.

[5] J. Botia. Debugging huge multi-agent systems: group and social perspectives. 2005.

[6] Rem W. Collier. Debugging agents in agent factory. In *Programming Multi-Agent Systems*, pages 229–248. Springer Berlin Heidelberg, 2007.

[7] Rem W. Collier, Seán Russell, and David Lillis. Reflecting on Agent Programming with AgentSpeak(L). In *Procedings of the 18th Conference on Principles and Practice of Multi-Agent Systems (PRIMA 2015)*, 2015.

[8] Mehdi Dastani. 2APL: a practical agent programming language. *Autonomous Agents and Multi-Agent Systems*, 16(3):214–248, 2008.

[9] Mehdi Dastani, M. van Birna Riemsdijk, and John-Jules Ch. Meyer. Programming multi-agent systems in 3APL. In *Multi-agent programming*, pages 39–67. Springer, 2005.

[10] Akshat Dhaon and Rem W. Collier. Multiple Inheritance in AgentSpeak (L)-Style Programming Languages. In *Proceedings of the 4th International Workshop on Programming based on Actors Agents & Decentralized Control*, pages 109–120. ACM, 2014.

[11] Dinh Doan Van Bien, David Lillis, and Rem W. Collier. Space-time diagram generation for profiling multi agent systems. In *Programming Multi-Agent Systems*, pages 170–184. Springer Berlin Heidelberg, 2010.

[12] Koen V. Hindriks. Programming Rational Agents in GOAL. In Amal El Fallah Seghrouchni, Jürgen Dix, Mehdi Dastani, and Rafael H Bordini, editors, *Multi-Agent Programming:*, pages 119–157. Springer US, 2009.

[13] Koen V. Hindriks. Debugging is explaining. In Iyad Rahwan, Wayne Wobcke, Sandip Sen, and Toshiharu Sugawara, editors, *PRIMA 2012: Principles and Practice of Multi-Agent Systems*, volume 7455 of *Lecture Notes in Computer Science*, pages 31–45. Springer Berlin Heidelberg, 2012.

[14] Koen V. Hindriks, Birna Van Riemsdijk, Tristan Behrens, Rien Korstanje, Nick Kraayenbrink, Wouter Pasman, and Lennard De Rijk. Unreal goal bots. In *Agents for games and simulations II*, pages 1–18. Springer, 2011.

[15] Nicholas R. Jennings. On agent-based software engineering. *Artificial intelligence*, 117(2):277–296, 2000.

[16] Dung N. Lam and K. Suzanne Barber. Debugging agent behavior in an implemented agent system. In Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah Seghrouchni, editors, *Programming Multi-Agent Systems*, volume 3346 of *Lecture Notes in Computer Science*, pages 104–125. Springer Berlin Heidelberg, 2005.

[17] David Lillis. *Internalising Interaction Protocols as First-Class Programming Elements in Multi Agent Systems*. PhD thesis, University College Dublin, 2012.

[18] Alessandro Ricci and Andrea Santi. Designing a General-purpose Programming Language Based on Agent-oriented Abstractions: The simpAL Project. In *Proceedings of the Compilation of the Co-located Workshops on DSM'11, TMC'11, AGERE! 2011, AOOPES'11, NEAT'11, & VMIL'11*, SPLASH '11 Workshops, pages 159–170, New York, NY, USA, 2011. ACM.

[19] Alessandro Ricci and Andrea Santi. Typing Multi-agent Programs in simpAL. In Mehdi Dastani, Jomi F. Hübner, and Brian Logan, editors, *Programming Multi-Agent Systems*, volume 7837 of *Lecture Notes in Computer Science*, pages 138–157. Springer Berlin Heidelberg, 2013.

[20] Alessandro Ricci, Mirko Viroli, and Andrea Omicini. CArtAgO: A Framework for Prototyping Artifact-Based Environments in MAS. In Danny Weyns, H. Van Dyke Parunak, and Fabien Michel, editors, *Environments for Multi-Agent Systems III*, volume 4389 of *Lecture Notes in Computer Science*, pages 67–86. Springer Berlin Heidelberg, 2007.

[21] Seán Russell. *Real-time monitoring and validation of waste transportation using intelligent agents and pattern recognition*. PhD thesis, University College Dublin, 2015.

[22] Seán Russell, Howell R. Jordan, G.M.P. O'Hare, and Rem W. Collier. Agent Factory: A Framework for Prototyping Logic-Based AOP Languages. In Franziska Klügl and Sascha Ossowski, editors, *Multiagent System Technologies*, volume 6973 of *Lecture Notes in Computer Science*, pages 125–136. Springer Berlin Heidelberg, 2011.

[23] Yoav Shoham. Agent-oriented programming. *Artificial intelligence*, 60(1):51–92, 1993.