Multiple Inheritance in AgentSpeak(L)-style Programming Languages

Akshat Dhaon
UCD School of Computer Science & Informatics,
University College Dublin, Belfield, Dublin 4, Ireland
akshat.dhaon@ucdconnect.ie

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

Abstract

Agent-Oriented Programming (AOP) is a high-level programming paradigm for implementing intelligent distributed systems. While a number of AOP languages have been proposed in the literature, many of them focus on the provision of support for intelligent decision making rather than addressing language design concerns such modularity and reuse. To address this imbalance, this paper presents an abstract model of multiple inheritance for AgentSpeak(L) style languages which decomposes agent programs into a set of inter-related agent classes and defines a run-time apparatus for rule selection on the relationship between those classes. To demonstrate our approach, a case study is presented that introduces a new AgentSpeak(L)-based language entitled ASTRA and its use in an illustrative implementation of an agent-based chat system is then presented.

Categories and Subject Descriptors D.3.3 [**Programming Languages**]: Language Constructs and Features – abstract data types, polymorphism, control structures.

General Terms Algorithms, Design, Languages, Theory.

Keywords Agent Programming, Multiple Inheritance.

1. Introduction

Agent-oriented programming is an upcoming programming paradigm that adopts a societal view of computation [28] in which systems are built from a set of entities known as *agents* that coordinate their activities in order to realise global system behaviours. Individual agents are imbued with intelligent decision making capabilities that allow them to autonomously determine when and how to interact with other agents in the system.

To date, a number of agent programming languages have been proposed [5] that support programming of the intelligent behaviour based on mental notions such as beliefs, desires, and intentions [25] [28]. Many of these languages are logic based and are inspired by earlier theories of rational action. Unfortunately, research on these languages has tended to focus on the decision-making capabilities of agents to the exclusion of other important issues, such as modularity and code reuse. This lack of consideration has led to a number of languages that can be applied to solve complex problems, but are rarely used because the implementations are difficult to understand, maintain and reuse.

Some previous attempts have been made to address the issue of modularity. For example, [8] [9] propose a modular approach to reuse using the concept of capabilities- reusable clusters of beliefs, plans and events - that can act as the building blocks of agent programs. Others have proposed the use of shared environments as reuse mechanisms, which allow heterogeneous agents to share common action and perception routines [19]. However, the provided modularity constructs are mostly compositional, and support for variability through specialisation, which has long been considered a key issue in mainstream programming languages, is less well explored [30]. Finally, in [19], a generalised model of single inheritance is proposed that could be applied to all agent programming languages. However, this model is intended to be applied as a pre-processing step that is lost at compile time. It also supports only overriding of behaviours and not extension of behaviours.

In our view, single inheritance does not provide an adequate model of specialisation for agent programming languages. This is apparent when attempting to utilize an agent programming language to implement a design specified in one of the many agent-oriented methodologies (e.g. GAIA [36], Prometheus [24]). Typically, these methodologies define system behaviours in terms of a set of roles that interact to achieve the required behaviour. These roles are then combined into a set of agent types that are to be implemented. This type of decomposition is used because it is generally seen as more natural and as offering a better form of decomposition than agent types. Unfortunately, due to the lack of a correlation between the key concept of a role and an equivalent concept in the agent programming language, implementations of the design rarely bear any resemblance to the design. While this has led some researchers to investigate the notion of roles as run-time constructs [12][16], there is no common consensus as to how roles should be realised in agent programming language: either as an equivalent to a Java interface for representing interaction; or as a concept that encapsulates partial behaviours of agents.

In methodologies, the final set of agent types that make up a system design often includes a number of agent types that are associated with multiple roles. Further, many of the roles are associated with multiple types. This relationship between roles and types has a strong resonance with multiple inheritance: roles represent specific behaviours in the system that can be implemented in individual agent classes which can, in turn, be combined into final agent classes that correspond to the required agent types.

SIGPLAN'05 June 12–15, 2005, Location, State, Country.

This paper explores the use of multiple inheritance in AgentSpeak(L) [25] style agent programming languages. We have focused on this class of languages because our model is not only a design-time construct, but also applied at run-time. However, we believe that the approach we have adopted can be applied to other agent programming languages as well.

2. Multiple Inheritance

Multiple inheritance is a powerful paradigm in object-oriented programming. It is of great interest in Artificial Intelligence for its classification power [7] [32] [33]. It is also a very promising programming tool with good properties for modularity, reusability and incremental design [13] [22]. There has been much discussion about the usefulness of multiple inheritance [6]. It is also a complex notion as defining and using or reusing non trivial hierarchies of classes needs precise principles and powerful mechanisms [10]. When a class is derived from exactly one parent (base) class, it is referred to as single inheritance while if a class is derived from more than one parent classes, it is multiple inheritance. Graphs for single inheritance are always trees. Graphs for multiple inheritance are DAGs (directed acyclic graphs). In general, multiple inheritance allows a user to combine independent (and not so independent) concepts represented as classes into a composite concept represented as a derived class [29].

There are two strategies in common use for dealing with multiple inheritance. The first strategy attempts to deal with the inheritance graph directly. This is consistent with the approach of Trellis/Owl, extended Smalltalk and C++. If a class inherits operations with the same name from more than one parent, the conflict must be resolved. Trellis/Owl deals with it by signalling an error at compile time. On the other hand, extended Smalltalk creates an operation for the class that signals an error when invoked [29]. However, if the same operation (from the same parent) is inherited by a class via different paths through the inheritance graph, it is not an error [29]. Thus conflict resolution is an integral part of this strategy. As in Smalltalk, a compound selector may be used to specify the parent implementation to be invoked [10]. Also, for cases where no compound selector is used and there is a need to provide a conflict resolution mechanism, a linearization strategy, like that discussed in [1], can be put in place. In this regard, acceptability and monotonicity are two of the most desirable characteristics for linearization [17]. Various strategies for the same have been studied [14].

The second strategy first flattens the graph into a linear chain, and then deals with that chain using the rules for single inheritance [29]. This strategy is used by Flavors [23] and CommonLoops [3]. A total ordering is created that preserves the ordering along each path through the inheritance graph (a class never appears after one of its ancestors). Flavors attempts to preserve the relative ordering of the parents of a class too (the first parent of a class never appears after the second, etc.).An error is signalled if no total ordering exists that satisfies all such constraints. While the relative ordering of a class and its ancestors is preserved in the total ordering, unrelated classes can be inserted between a class and its parent. The computed inheritance chain has the unusual property that the "effective parent" of a class x may be a class of which the designer of class x has absolutely no knowledge of [29]. For example if developer 1 designs classes x & y and developer 2 designs z & w (x inherits y and z inherits w) and there are some more inheritances involved in the system, there may be a case where, in the flattened chain, z is the parent of x. While there would be no issues at compile time in such cases, they may result in anomalous run-time behaviours. One more problem with the computed chain is that in case of operation conflicts (two or more parents defining the same operation) one operation will be selected even if there is no clear "best" choice [29]. Another problem involves the way in which a class may reliably communicate with its "real" parents [29]. CommonLoops allows a class to invoke its "effective parent" (its parent in the computed inheritance chain) using a notation similar to super in Smalltalk. Flavors provides a declarative mechanism (method combination) that is essentially equivalent. These mechanisms avoid the problem of unintentional multiple invocations of operations that can occur using graph-oriented solutions. However, these mechanisms make it difficult for a class to reliably set up private communication with a parent, given the interleaving of classes that can occur when the inheritance chain is computed [29].

Recently, automated delegation [34], mixins [20] and traits [27] have come up as ways of reuse in the agent programming community. However, they are able to provide more of an extension of behaviours but not combination. They can only be considered as alternative ways of composition and can't address issues like that of independent control (if an agent is autonomous, how can it make decisions involving conflicts or how would an agent be controlled in case an organizational structure is implemented).

3. Our Solution

In this paper, we have chosen to demonstrate how multiple inheritance can be applied to AgentSpeak-style languages. As was discussed in the introduction, we have done this because our model impacts the design of the interpreter. AgentSpeak has been chosen because it represents the most prominent class of agent programming languages and is the basis for the highly successful Jason agent programming language [4].

At its core, AgentSpeak(L) is an event driven language in which the agent processes events relating to its internal decision making (goal events) or its external environment (belief update events). One event is processed per iteration, and the event is handled by selecting a plan rule. Plan rules define courses of action (plans) that should be used to in response to the occurrence of specific events in a given context (state of the environment).

Traditionally, an AgentSpeak(L) agent consists of an event queue, a set of beliefs (representing the current state of the environment), a set of plan rules (the behaviour of the agent) and a set of intentions (the current active behaviours being executed by the agent). A central feature of the underlying interpreter is the algorithm that selects which plan rule should be used to handle a given event. In most implementations, this algorithm checks rules in the order in which they are written (in the agent program source code) and selects the first rule that handles the given event and whose context is satisfied.

In our model, we introduce the notion of an agent class as a container for plan rules. This allows us to decompose the monolithic list of plan rules typically attributed to agent program into coherent lists of plan rules that correspond to specific behaviours. These agent classes can then be combined through multiple inheritance to define composite behaviours. An agent program is then defined as an instance of an agent class.

The consequence of introducing the concept of an agent class is that the plan rule selection algorithm is no longer sufficient, especially in the case where a behaviour is split over multiple agent classes. In this scenario, the key question we must ask is *how can we determine which lists of rules should be checked and in what order?*

In this way, rule selection is a similar problem to the issues of method selection in Object-Oriented Programming languages that employ multiple-inheritance. This issue was discussed in some detail in section 2. Out of the strategies outlined that section, our model adopts the first strategy - it deals with the inheritance graph directly. However, we also maintain a linearized chain as part of a conflict resolution strategy. Informally, the conflict resolution strategy adopted is as follows:-

- If a rule that appears in multiple parents is invoked in a child, the invocation of the correct sequence of instructions would depend on the order of inheritance by default, i.e. the corresponding rule of the class inherited first is executed, if present in it. Otherwise the next inherited class would be checked, and so on.
- The scope resolution operator ("`::`"), similar to that of C++, can be used to explicitly define the scope of invoking a rule if it is present in multiple agents of the hierarchy.
- If no scope is defined while invoking a rule in the child agent and it is present in one or more of its parents as well as the child itself, the implementation/s defined in the parent/s would be hidden by that of the child (overriding).

To illustrate and refine the model in more detail, this section presents an informal overview of the modified AgentSpeak(L) language, followed by a number of examples that illustrate key features of the model.

3.1 Modified AgentSpeak(L) Grammar

We modify and extend the basic AgentSpeak(L) syntax to include a unique identifier for each agent program (the agent class name), a list of the agent classes that are being extended, and introduce the concept of a *scoped goal* as a type of sub-goal where the corresponding events are targeted at a specific agent class.

A Java style syntax has been adopted for this extension, however any appropriate syntax could be used. In particular, the "agent" keyword is used to denote an agent class and the "extends" keyword to specify the parent agent class(es). Multiple parent classes can be specified using a comma-delimited list of agent class names. The body of the agent program (any initial state and rules) is enclosed within a set of braces.

The syntax for these proposed changes are illustrated in Figure 1 with the italicised syntax represents the extensions proposed to the existing language.

```
          A  ::=  [agent <agent-name>
                  [extends <agent-name> (,
                  <agent-name>)*]<c>*
          C  ::=  <initial> | <rule>
    initial  ::=  <belief> | <goal>
       rule  ::=  <event>: <context><-
                  (<statement>)*
  statement  ::=  ? <belief> | + <belief>
                  | - <belief> | <goal> |
                  <action> | <scoped-goal>
scoped-goal  ::=  <agent-name> :: <goal>
```

Figure 1. Modified AgentSpeak(L) syntax

3.2 Combining Behaviours

The most obvious benefit of multiple inheritance is the ability to combine multiple existing behaviours to form a new behaviour. For example, in the scenario shown in Figure 2, agent A has a rule that handles the event corresponding to the addition of the goal `!init()` while the agent B has two rules that correspond to the addition of two goals, `!init()` and `!myInit()`. Agent C, which is a child of both A & B according to our model, will now be able to invoke both the goals even though it itself doesn't define either of them. This is how we can combine the behaviours of multiple agents into a new agent using multiple inheritance.

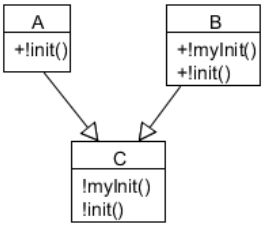

Figure 2. A simple multiple inheritance scenario

If we look at the scenario a little more closely, we can infer that calling `!myInit()` in C would invoke the `!myInit()` defined in B. However, it is not clear what would happen when we call `!init()` in C.There are two possibilities, either the `!init()` of A is invoked or that of B is invoked. Such a problem has also been encountered in object-oriented programming (in languages like C++), when a method called from a child instance exists in multiple parents. C++ deals with it by giving a compile time error as the method referenced is ambiguous. However, C++ also provides the "virtual" keyword which enables a programmer to bypass the compilation error. If the method existing in multiple parent classes is declared virtual and referenced from the child instance, then the implementation defined by the parent inherited first (i.e. appearing on the left in the line where parents of the child class are defined) would be invoked.

Rather than present compile time errors, this is the default behaviour adopted in our model, i.e. if a rule defined in multiple parents is invoked from the child agent, the implementation defined by the parent appearing on the left in the "extends" line gets preference and is called by default. This means that the `!init()` of A would be invoked in this case.

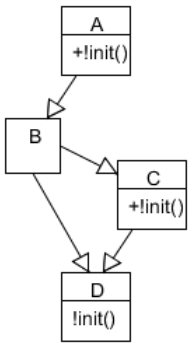

Figure 3. A more complex multiple inheritance scenario

3.3 Overriding Behaviour

A second benefit of multiple (and single) inheritance is the ability to override existing behaviours, to provide a modified behaviour. For example, in Figure 3 agent A has a rule that handles the event corresponding to the addition of the goal !init() while agent B inherits A. Agent B, in turn, is inherited by C, which again adds a rule that handles the event corresponding to the addition of the goal !init(). Finally, agent D inherits B and C and tries to invoke the !init() implementation.

It is to be noted here that the !init() implementation defined in C would hide (override) the implementation that had been defined in A as C is lower down in the hierarchy. However D would also get A's implementation of !init() through B. This is a little more complicated than the case discussed in section 3.2 as the choice is not obvious. But, in our implementation, we first give preference to a smaller distance from root (the agent from which the rule is being invoked). Hence invoking !init() on D would invoke the implementation defined in C and not in A. The case discussed in section 3.2 would apply only when distances from root for two parent agents are the same.

3.4 Scope Resolution

Sometimes the default conflict resolution strategy outlined in 3.3 can lead to scenarios in which the selected rule is not the one we desire. For example -again referring to Figure 3 - what if we need to invoke A's implementation of `!init()` from D? For such a scenario, we introduce the scope resolution operator ("**::**"), similar to that provided by C++. This can be used to restrict the scope of a construct invocation. For the requirement discussed above, we can use `A::!init()`. It should be noted that the same implementation of A can also be invoked by `B::!init()` as when we restrict the scope of `init()` to B, the only implementation available in the hierarchy is that of A.

In this context, let us also look at Figure 2 again. What if a programmer needs to invoke the `!init()` of B from C? For such a scenario also, `B::!init()` would be useful. Therefore, the scope operator can always be used by the programmer to specify the scope of the construct to be invoked. In cases where it is not used, default behaviour is used to avoid ambiguities.

3.5 Implications

In order to illustrate the potential benefits and limitations of our model, this section explores some examples of different ways in which it can be used in real world systems.

3.5.1 Extending Behaviours

```
agent A {
+!init() <-
   println("Hi from A");
}

agent B extends A {
+!init() <-
   println("Hi from B");
   A::!init();
}
```

Figure 4. Example for Extending Behaviours

Figure 4 depicts how B inherits `!init()` from A and extends the behaviour associated with the `!init()` goal. This is how the inheritance model may be used to extend existing behaviours.

3.5.2 Combining Behaviours

```
agent A {
+!init() <-
   println("Hi from A");
}

agent B {
+!init() <-
   println("Hi from B");
}

agent C extends A, B {
+!init() <-
   A::!init();
   B::!init();
}
```

Figure 5. Example for Combining Behaviours

Figure 5 depicts how C inherits `!init()` from A and B and combines the existing behaviour of both to produce a new behaviour for `!init()`.

3.5.3 Defensive Programming

```
agent A {
count(0);

+!init() : count(x) & x<50<-
   A::!inc();
   //Do something
   !init();

+!inc() : count(x) <-
   -count(x);
   +count(x+1);
}

agent B extends A {
+!inc() : count(x) <-
   -count(x);
   +count(x-1);
}

agent C extends A, B {
   !init();
}
```

Figure 6. A Defensive Programming Example

The ability to extend behaviour can also introduce unexpected side-effects. For example, in Figure 6, the code has been written to do something 50 times (in agent A). However, it is possible to extend that agent to override the correct functionality with something incorrect and cause bugs in the code. To illustrate this, we show agent B overriding the `!inc()` goal of A to decrement the count, instead of incrementing it, which would result in incorrect behaviour. Fortunately, the scope operator helps us to avoid this undesirable outcome. By writing `A::!inc()`, the developer ensures that his code won't be overridden and hence never have an incorrect behaviour. Thus we see how there can be a need to limit overriding in certain cases to oversee the dangers that can be caused by our model. Now, whenever `!init()` will be called from agent C, it would always give the correct behaviour. This can be considered analogous to what the "final" keyword achieves in OOP languages like C++ and Java. However, while final is a compile-time feature, the feature of restricting scope is at run-time. This implies that it can provide

more flexibility than *final*. B could still override the `!inc()`of A and C (or any agent directly calling `!inc()`) can choose whether to use the overridden version or not. In case where an agent utilises a goal/plan of its parent indirectly, the parent is able to restrict or allow overriding (as shown here).

4. Formalisation

This section presents a formal model that illustrates the semantics of the modified language. Our model is inspired by the Jason formal model presented in [17]. However, we do not attempt to provide a complete formalisation of the language, but instead we choose to focus only on the relevant bits (which need to be modified for our purpose). Specifically, we focus on the revised plan selection mechanism that is necessary to cater for the introduction of multiple inheritance.

The key concepts are as follows:-

- An agent's class is defined as

  `Class = <name, P, IS, R>` where:

  - `name` is the name of the class.
  - `P` is the list of parents of the class.
  - `IS` is the initial state of the class.
  - `R` is an ordered list of rules associated with the class.

- An agent is defined as

  `Agent = <name, type, B, I, E>` where:

  - `name` is the agent identifier.
  - `type` is the name of the type (class) of the agent.
  - `B` is the set of beliefs that the agent has.
  - `I` is the set of intentions which the agent has.
  - `E` is the event queue associated with the agent.

- An event is defined as

  `Event = <te, i, type>` where:

  - `te` is the triggering event.
  - `i` is the source of the event (an intention or ϕ meaning it is an external source).
  - `type` is the name of the scoped class or ϕ if not scoped.

- An agent program is defined as

  `Program = <C, A>` where:

  - `C` is a set of agent classes associated with the program.
  - `A` is a set of agents associated with the program.

Apart from introducing the concept of an agent class, we have modified AgentSpeak(L) events to cater to scoping. In this new model, rules are linked to classes and the adoption of multiple inheritance impacts rule selection. We now proceed to provide the algorithms that would determine rule selection in the modified language.

```
execute(P, A)
   e <- selectEvent(A)
   o <- selectOption(P, A, e)
   if (o !=ϕ) then
        applyOption(o)
   endif
   i<- selectIntention(A)
   if (i != ϕ) then
        executeIntention(i)
   endif
```

Figure 7. Simplified AgentSpeak(L) Interpreter Algorithm

Figure 7 is a simplified version of the AgentSpeak(L) interpreter algorithm presented in [26]. In our model, only the italicised line (`selectOption`) is affected. While it is simpler than the algorithms proposed in [26] [17], it does correspond to those algorithms.

```
selectOption(P, A, e)
   if e.type = ϕ then
        t <- A.type
   else
        t <- e.type
   endif
   classes <- getLinearization(P, t)
   while (classes != []) do
        cls <- head(classes)
        classes <- tail(classes)
        o <- selectOptionForClass(P,A,cls,e)
        if (o != ϕ) then
          return o
       endif
   endwhile
   return ϕ
```

Figure 8. ModifiedAgentSpeak(L) selectOption Algorithm

Figure 8 shows our modification to the existing `selectOption` algorithm of AgentSpeak(L). The initial part of our algorithm deals with the handling of scope (discussed in section 3.4 above). If no scope is defined by the user, then the scope would default to the current agent class. We then introduce a `getLinearization` algorithm which returns a list of classes according to the criteria discussed in sections 3.2 & 3.3. We do not present the whole algorithm here for brevity constraints. The `selectOptionForClass` works in exactly the same way as the existing AgentSpeak(L) `selectOption` algorithm, there is an extra parameter for class as this algorithm can be called multiple times (once for each agent class in the hierarchy defined by the scope).

5. Case Study

We present a case study where our model is applied to a recently developed agent programming language, called ASTRA [11]. ASTRA is an agent-oriented programming language that is built on Java. It is an implementation of AgentSpeak(TR), which extends AgentSpeak(L) with support for Teleo-Reactive functions [11].

ASTRA is designed to be syntactically close to Java. The programs are written in files with a ".astra" extension. Each file must contain exactly one agent class, specified by the **agent** keyword. In terms of this paper, the important changes features are:-

- Initial beliefs & goals can be specified using the **initial** keyword.

```
package chat;
agent Manager {
rule @message(request,string S, connect(string UN)):user(S,string SN) {
      if (~user(UN,string N)) {
         send(refuse,S,connect(UN));
      } else {
         send(agree,S,connect(UN));
         query(user(UN,string name));
         send(propose,UN,connection(S, SN));
         when(connect_response(UN,S,SN, string R)) {
               -connect_response(UN,S,SN, R);
         if (R == "yes") {
               send(inform,S, connection(UN,name));
               } else {
               send(failure,S, connect(UN));
               }
         }
      }
   }

   rule @message(accept-proposal, string S, connection(string UN,string N)) {
      +connect_response(S,UN,N,"yes");
   }

   rule @message(reject-proposal, string S, connection(string UN,string N)) {
      +connect_response(S,UN,N,"no");
   }
}
```

Figure 9. Code for ASTRA Chat System Manager

- Plan rules are specified using the **rule** keyword.
- Standard statement types for plans are provided: if, while, foreach, variable declarations, assignment, try...recover etc.
- Support for system and user defined libraries of actions and sensors that are implemented as annotated java classes, known as **modules**.
- Provision of integrated support for CArtAgo [26] and EIS [2].
- Introduction of a type system for variables that is designed to map on to java primitives (i.e. int, float, long, double, char etc.) and objects.
- Additional event types are introduced for messages, and EIS/CARTAGO events.

Due to space constraints, details of the ASTRA language are not presented here. Instead, the reader is directed to the ASTRA language website. Some examples of ASTRA code are presented in Figures 9, 10 and 11.

5.1 Example System without Inheritance

Here we present some code from a chat system implemented in ASTRA. This system was first written without using inheritance. It was presented as part of the *Agent Oriented Software* module of the course, *M.Sc. Advanced Software Engineering* 2013-14 at *University College Dublin.*

The code presented in Figure 9 is for the Manager which manages all connection requests (between various users) in the system. The normal behaviour of the Manager agent is that it responds to an initial request from a User agent to connect to another user (the first rule in Figure 9). If the Manager believes that the connecting agent is a registered user of the system then it checks if the target user exists. If the target user does not exist, the Manager refuses the connection request. If the target user does exist, then the Manager agrees to the connection requests and sends a propose message to the target agent asking whether or not the connection can be made. The plan then waits for a response to be returned.

Receipt of the response to the proposal is handled by the second and third rules which deal with acceptance and rejection of the proposal respectively. When the Manager receives its response, it adopts a belief that causes the main plan to stop waiting and to continue processing the connection request. If the proposal was accepted, it informs the requesting User agent of the success of the initial request, otherwise it tells the User agent that its connection request has failed.

In this example, which represents typical a typical solution to this type of scenario, we can see how protocols are implemented in a bespoke way to handle the different messages that can be sent in the system. This is a very basic approach where each step in the protocols has to be coded explicitly in each agent. It should be noted that there is not much scope of code reuse in this approach. So when we write code for other entities in the system (e.g. User), we do it again from scratch.

5.2 Example System with Inheritance

We now present code for the same system where we utilise multiple inheritance. First of all, we code the various protocols that would be used by different entities in the same system as well as other systems. Thus we build a library of standard interaction protocols, the fipa protocols [18].

In Figure 10, we present the Propose protocol. It is the simplest of all fipa protocols. This protocol can be utilised by an agent that wishes to initiate a Propose communication or by an agent that is supposed to participate in it. To initiate this type of

```
package astra.fipa.protocol;
agent Propose extends Protocol {
   // ------------------------------
   // Initiator
   // ------------------------------
   rule +!fipa_propose(int T, string S, string A, list P, boolean R) {
        !start_conv("fipaPropose",int Id);

        send(propose, sender, content("fipaPropose",Id,A,P));

        !!fipa_timeout(Id, T);
        wait(fipa_timeout(cid) | fipa_result(Id,boolean R2));

        if (fipa_result(Id, boolean B)) {
          R = B;
          -fipa_result(Id, B);
        } else {
          R = false;
        }
   }

   rule @message(accept-proposal, string S, content("fipaPropose", int Id)) {
        +fipa_result(Id, true);
   }

   rule @message(reject-proposal,string S, content("fipaPropose", int Id)) {
        +fipa_result(Id, false);
   }

   // ------------------------------
   // Participant
   // ------------------------------
   rule @message(propose, string S, content("fipaPropose", int Id, string A,
   listP)) {
      !handle_fipa_propose(S,A,P, boolean R);
      if (R == true) {
         send(accept-proposal, S, content("fipaPropose", Id));
      } else {
         send(reject-proposal, S, content("fipaPropose", Id));
      }
   }

   rule +!handle_fipa_propose(string S, string A, list P, boolean R) {
      R = false;
   }
}
```

Figure 10. Code for the Propose Protocol

communication, an initiator first creates a conversation and then sends a propose message to the agent that should participate. It also adopts a timeout sub-goal implying that it would wait for the response of this message for finite time duration. Receipt of the response to the propose message sent by initiator is handled by the second and third rules. When a possible participant receives a propose message, it can choose to either accept or reject the proposal. The rules for this are in the participant section of Figure 10.

Due to space constraints, we won't be able to show the other protocols (such as Request and Subscribe) here. However, all protocols could be written along the lines of the Propose protocol.

The Protocol class, present in the system, works as a building block for various protocols. It contains goals such as `start_conv` and `fipa_timeout` as well as the mechanism for generating unique conversation ids. The agents that extend this class do not have to worry about the implementation details of these goals.

Figure 11 presents the code for a Manager which handles all connection requests in the system as well as utilises the features of multiple inheritance. It should be noted that this code looks quite concise with the Manager only needing to handle rules relevant to the conversation without bothering about the details of the underlying Request and Propose protocols.

The whole code (including that in Figure 10) may not be lesser than its counterpart that doesn't use inheritance if we consider the lines of code (LoC) metric. However, we see how protocols like Propose are decoupled from the business logic. These protocols may be reused by other entities in the system (e.g. User) and by entities in other systems as well. Implementation details of agents at one level of the hierarchy are abstracted from the other levels. Hence, agents at each level need to focus only on their core functionalities thus enabling an incremental development of the system. It was also possible for us to add another feature, that of timeout handling, in this scenario with minimal effort. Similarly, we could also have added the logic of

```
packageastra.fipa;
agent Manager extends Request,Propose {
   rule +!handle_fipa_request(string S, "connect", list P, boolean R) {
      if (~user(S, prelude.valueAsString(P,0))) {
         result = false;
      } else {
         result = true;
      }
   }

   rule +!process_fipa_request(string S, "connect", list P, list V): user(S, string SN) {
      query(user(prelude.valueAsString(P,0),string N));
      !fipa_propose(UN, "connection", [S,SN], boolean R);
      if (result == true) {
         V = [N];
      } else {
         system.fail();
   }
}
```

Figure 11. Code for the ASTRA Chat System Manager using Multiple Inheritance

cancelling a conversation. As a result, we see that multiple inheritance allows us to build a more concrete, robust and extensible model. Also, as different entities and protocols are decoupled, the system becomes much simpler to develop and comprehend. Coming back to the LoC argument, if we imagine a lot of entities present in the system, then the LoC can be much less for a multiple inheritance implementation as repetitive code would be written only once.

In this context, it must be noted that in the previous version of Manager that we had (which didn't use inheritance), the business logic was entwined with the protocols. It was a blurred, ad hoc solution that was not able to provide any consistency of approach or separation of concerns. Common issues arising out of using such a coding approach include the mismatch of performatives between the content & predicate of messages and mismatch in the type and number of parameters. Also, trying to add new features would have needed considerable effort. However, these problems were overcome with the new version (which utilised multiple inheritance) and we moved towards better coding practices. We were able to provide a better quality of code in the new version as we reused well-tested code implementations. This was a more engineered solution. Adopting such practices could lead to rapid and iterative development of software with shorter development cycles.

6. Conclusion

We conclude that multiple inheritance makes it simpler to implement a large, complex system. Also, different functionalities (roles) can be developed independently and combined later. The development can be iterative and decoupling of business logic implies that the focus can be on one feature at a time. Alternatively, many developers can be involved in the parallel development of a system, with each of them focusing on one feature. Multiple inheritance allows an agent to have more than one roles in a system, thus making them closer to the real world. Different responsibilities, pertaining to different roles, may be executed in different contexts by the same agent (rule selection). Regarding the LoC metric, the larger and more complex a system, the better would be its LoC as compared to its counterpart that does not utilise inheritance.

It should be noted that initially we had considered using only a single inheritance model. However, a number of factors led us towards adopting the multiple inheritance strategy. Single inheritance would allow us to extend single behaviours but not allow combining different behaviours. We felt that multiple inheritance was a natural fit for agent programming as it would allow agents to combine social and individual behaviours. Established agent methodologies like GAIA, AgentFactory, Prometheus etc. deal with identifying roles in a system. Once the roles are identified, the implementation of the modelled system would include implementing the roles into agents. This served as another motivation for multiple inheritance so that one agent had the capability to combine multiple roles. Thus we can say that multiple inheritance fits nicely with existing Agent Oriented Software Engineering methodologies. Also, multiple inheritance can allow easy encoding of market patterns such as Product Hierarchy, Functional Hierarchy, Centralised Market etc. into agents. In addition to this, multiple inheritance corresponds to the roadmap described in [21].

Through our findings, we can say that multiple inheritance improves an agent language by:-

- Modularising it.
- Give the ability to reuse complex agent programs (rules, plans, and goals).
- Provide customisable templates to users to reduce development times.

Each agent may be a combination of more than one template, utilising some/all of the features defined in earlier templates without needing to have them at a single place and/or redefining them.

In our chosen case study, the motivating factor for multiple inheritance was the ability of an agent to combine the implementation of multiple protocols. As our new Manager agent combines both Request and Propose protocols, it has the ability to exhibit both kinds of behaviour (based on different contexts). Also, established protocols can be reused by different agents in the same system or even in other systems. Such a level of ab-

straction would aid in reducing complexity. With agents already established as smart entities, such abilities would open new possibilities for software development.

References

1. Barrett, Cassels, Haahr, Moon, Playford & Withington. “A Monotonic Superclass Linearization for Dylan.” In OOPSLA '96 Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, Pages 69-82.
2. Behrens, T., Dix, J., Hindricks, K. V., “Towards an environment interface standard for agent platforms”, Annals of Mathematics and Artificial Intelligence, Volume 61, Issue 4, pp 261-295, April 2011
3. Bobrow, D., Khan, K., Kiczales, G., Masinter, L., Stefik, M., Zdybel, F., “CommonLoops: Merging Common Lisp and Object-Oriented Programming.” Proc. ACM Conference on Object-Oriented Systems, Languages, and Applications. Portland, Oregon, Sept. 1986.
4. Bordini, Hübner & Renata. “Jason and the Golden Fleece of Agent-Oriented Programming.” In Multiagent Systems, Artificial Societies, and Simulated Organizations Volume 15, 2005.
5. Bordini, R., Braubach, L., Dastani, M., Seghrouchni, A, Gomez-Sanz, Leite, J., O’Hare, G.M.P., Pokahr A. & Ricci, A. “A Survey of Programming Languages and Platforms for Multi-Agent Systems.” Informatica 30 (2006) 33–44.
6. Borning & Ingalls. “Multiple Inheritance in Smalltalk-80.” In ECOOP '87. European Conference on Object-Oriented Programming: Paris, France.
7. Brachman, R. J. “What IS-A is and Isn’t: An Analysis of Taxonomic Links in Semantic Networks,” IEEE Computer, vol. 16, no. 10, 1983.
8. Braubach, Pokahr, & Lamersdorf. “Extending the capability concept for flexible BDI agent modularization.” Programming Multi-Agent Systems, pages 139–155, 2006.
9. Busetta, Howden, Ronnquist, & Hodgson. “Structuring BDI agents in functional clusters.” Intelligent Agents VI. Agent Theories Architectures, and Languages, pages 277–289, 2000.
10. Carre, B., Geib, J-M., “The Point of View notion for Multiple Inheritance”.In OOPSLA/ECOOP '90 Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications, Pages 312-321
11. Collier, R.W., “ASTRA Language Website”, URL: http://www.astralanguage.com/, 2014
12. Collier, R., Ross, R., O'Hare, G.M.P., A Role-based Approach to Reuse in Agent-Oriented Programming, AAAI Fall Symposium on Roles, an interdisciplinary perspective (Roles 2005), November 3-6, Hyatt Crystal City, Arlington, Virginia, USA, 2005
13. Cox, B.J. “Object-Oriented Programming: An Evolutionary Approach”, Addison-Wesley, Reading (Mass.), 1986.
14. Crespo, J., Marques J. M., Rodriguez, J.J. “On the Translation of Multiple Inheritance Hierarchies into Single Inheritance Hierarchies.” In Proceedings of the Inheritance Workshop at ECOOP 2002, Publications of Information Technology Research Institute, 12/2002, University of Jyvskyl.
15. Daly, J., Brooks, A., Miller, J., Roper M., & Wood, M.. “The effect of inheritance on the maintainability of object-oriented software: an empirical study.” In Proceedings of the International Conference on Software Maintenance, pages 20–29. IEEE, 1995.
16. Dastani, M., Birna van Riems-dijk, M., Hulstijn, J. Dignum, F., Ch. Meyer, J. “Enacting and deacting roles in agent programming”, In Proceedings of the 2nd International Workshop on Programming Multi-Agent Systems PROMAS2004), 2004.
17. Ducournau, Habib, Huchard & Mugnier. “Proposal for a Monotonic Multiple Inheritance Linearization.” In ACM SIGPLAN Notices, 01/1994; 29(10):164-175.
18. FIPA, “FIPA 2000 Specifications”, http://www.fipa.org/specs/, 2000
19. Jordan, H.R., Russell, S.E., O’Hare, G.M.P., Collier. R.W. “Reuse by Inheritance in Agent Programming Languages.” University College Dublin. In Proceedings of the 5th International Symposium on Intelligent Distributed Computing - IDC 2011, Delft, The Netherlands, October 2011.
20. Limberghen, M.V., Mens. T. “Encapsulation and composition as orthogonal operators on mixins: a solution to multiple inheritance problems.” Object Oriented Systems (1996).
21. Luck, M., McBurney, P., Shehory O., Willmott, S., Agent Technology: Computing as Interaction (A Roadmap for Agent Based Computing), AgentLink, 2005. ISBN 085432 845 9
22. Meyer, B. “Object-Oriented Software Construction”, Prentice Hall, 1988.
23. Moon, D., “Object-Oriented Progamming with Flavors”. Proc. ACM Conference on Object-Oriented Systems, Languages, and Applications. Portland, Oregon, Sept. 1986.
24. Padgham, L., Winikoff. M. “The Prometheus Methodology”. In Methodologies and Software Engineering for Agent Systems, Multiagent Systems, Artificial Societies and Simulated Organizations Volume 11, 2004, pp 217-234.
25. Rao, A. “AgentSpeak(L): BDI Agents speak out in a logical computable language”. Australian Artificial Intelligence Institute, Melbourne, Australia. In MAAMAW '96 Proceedings of the 7th European workshop on Modelling autonomous agents in a multi-agent world: Pages 42-55.
26. Ricci, A., Viroli, M., Omicini, A., CArtAgO: A Framework for Prototyping Artifact-Based Environments in

MAS, Environments for MultiAgent Systems III, Lecture Notes in Computer Science 4389, May-June 2007
27. Scharli, Ducasse, Nierstrasz & Black. “Traits: Composable Units of Behaviour.” In the European Conference on Object-Oriented Programming (ECOOP) 2003.
28. Shoham, Y. “Agent-oriented programming”, Robotics Laboratory, Computer Science Department, Stanford University, USA. Artificial Intelligence 60 (1993) 51-92.
29. Snyder, A. “Encapsulation and Inheritance in Object-Oriented Programming Languages”, Hewlett-Packard Laboratories. CA, USA. In OOPLSA '86 Conference proceedings on Object-oriented programming systems, languages and applications, Pages 38-45
30. Stroustrup, B., “Multiple Inheritance for C++”, AT&T Bell Laboratories, 1999.
31. Svahnberg, Van Gurp, & Bosch. “A taxonomy of variability realization techniques.” Software: Practice and Experience, 35(8):705–754, 2005.
32. Tarr, Ossher, Harrison, &Sutton Jr. “N degrees of separation: multi-dimensional separation of concerns.” In Proceedings of the 21st international conference on Software engineering, pages 107–119. ACM, 1999.
33. Touretzky, D.S. “The Mathematics of Inheritance Systems”, Pitman, London, 1986.
34. Viega, Tutt & Behrends. “Automated Delegation is a Viable Alternative to Multiple Inheritance in Class Based Languages.”, UVA Technical Report CS-98-03, March 12, 1998
35. Wegner, P. “Classification in Object Oriented Systems”. ACM SIGPLAN Notice, vol. 21, no. 10, 1986.
36. Woolridge, M., Jennings, N. & Kinny D.. “The Gaia Methodology for Agent-Oriented Analysis and Design.” In Autonomous Agents and Multi-Agent Systems, September 2000, Volume 3, Issue 3, pp 285-312.