A Role-based Approach to Reuse in Agent-Oriented Programming

R. Collier[1] R. Ross[2] G.M.P. O'Hare[1]

[1]School of Computer Science and Informatics,
University College Dublin, Ireland
{rem.collier, gregory.ohare}@ucd.ie

[2]Department of Computer Science,
Universität Bremen, Germany
robertr@informatik.uni-bremen.de

Abstract

This paper describes an extension to the ALPHA (A Language for Programming Hybrid Agents) programming language that employs roles as run-time constructs. Specifically, this paper describes how the inclusion of this concept has facilitated the use of a number of OOP-based reuse mechanisms within the language. Finally, we illustrate the new version of ALPHA through a simple auction-based example.

Introduction

This paper describes an extension to the ALPHA (A Language for Programming Hybrid Agents) programming language that employs roles as run-time constructs. Specifically, this paper describes how the inclusion of this concept has facilitated the use of a number of OOP-based reuse mechanisms within the language. Finally, we illustrate the new version of ALPHA through a simple auction-based example.

One exception to this is the approach presented in (Karageorgos, Thompson and Mehandjiev 2003), which employs the notion of a role at implementation time to provide a reusable library of partial agent programs, known as roles, can be reused by Zeus developers. A key feature of their approach is the inclusion of a role specialization (which the authors acknowledge is similar to inheritance in OOP) mechanism that allows developers to extend existing roles to include new features.

Another role-based approach that employs the concept of a role at both implementation and run -time is *RoleEP* (Ubayashi and Tamai 2000), a role-based evolutionary programming environment for mobile agent systems that uses the concept of a role to represent related travelling/collaboration tasks. RoleEP engenders reuse through the representation of roles as Java classes to which agents bind dynamically at run-time.

Agent-Oriented Programming (AOP) (Shoham 1993) represents an alternative approach to programming agents through the use of high-level languages whose syntax and semantics are derived from models of agents as mental entities that reason about how best to act through a mental state architecture. Some of the more prominent AOP languages include 3APL (Dastani et al. 2003), AgentSpeak(L) (Rao, 1996), and Nuin (Dickinson and Wooldridge, 2003)

Recently work in this area has begun to explore how the concept of a role can be applied to AOP languages (Dastani et al. 2004). Specifically, the authors describe how the pre-existing 3APL programming language (Dastani et al., 2003) has been extended to support roles. Underpinning this extension is a formal model of a role that combines information received, objectives, and rules that define conditional norms and obligations. This model is used first to motivate the design of a role-playing agent, whose structure is formally specified, and then to drive the design of a revised agent interpreter. However, the work does not consider the issue of reuse.

In this paper, we build on this work by exploring how Object-Oriented Programming (OOP) reuse mechanisms such as inheritance, composition, and aggregation can be applied to AOP. A key difference between the approach presented here, and that presented in (Dastani et al. 2004) arises from their specification, which seems to limit an agent to only one active role at a time. Once activated, this role exclusively drives the agents subsequent behaviour, and will continue to do so until it is deactivated and another role is activated. This is in contrast with the more widely accepted view that an agent will potentially have (1) many activated roles at any given instant in time (Odell, Van Dyke Parunak and Bauer 2001), and (2) some of those activated roles may be different instantiations of the same role. For example, it is perfectly acceptable that an auctioneer agent should be able to auction two items simultaneously, as often happens in the context of property sales.

The ability of an agent to play the same role many times at a given instance requires that the agent be able to distinguish between each occurrence of the role. In our property auctioneer example, it is possible to distinguish each occurrence of the role by the property that the auctioneer is selling. (Odell et al., 2003) discusses a related problem in the context of reusable Agent Unified Modelling Language (UML) Sequence Diagrams. The problem discussed arises when a specific sequence of agent interactions occurs repeatedly. Their solution to this problem is to model the repeated sequence of interactions as a separate protocol that has been generalised as a

template, which is then instantiated for each situation in which the common protocol is required.

This paper presents details of how a role-based approach to reuse has been engendered in the ALPHA programming language (Ross, Collier and O'Hare 2004), which sits at the heart of the Agent Factory framework (Collier 2001) (Collier et al. 2003). The approach adopted in this paper is based on practical experience gained from the use of the ALPHA programming language, and its predecessor, AF-APL (Collier 2001), in the development of a number of real world application domains (O'Hare and O'Grady 2003) (Muldoon et al. 2003).

ALPHA - A Language for Programming Hybrid Agents

ALPHA is an agent programming language that supports the development of agents that use a mental state architecture to reason about how best to act. Due to space constraints, only a brief summary of ALPHA is presented here. For an informal overview of the syntax and semantics of the language, the author is directed to (Ross, Collier and O'Hare 2004), and for a detailed overview of the logic that underpins the syntax and semantics of an earlier version of this language, known as AF-APL, the reader is directed to (Collier 2001).

ALPHA supports the fabrication of agents whose mental state is comprised of *beliefs*, *goals*, and *commitments*. Beliefs describe - possibly incorrectly - the state of the environment in which the agent is situated, goals describe future states of the environment that the agent would like to bring about, and commitments describe the activity that the agent is committed to realising. The behaviour of the agent is realised primarily through a purpose-built execution algorithm that is centred about the notion of *commitment management* (Collier 2001).

Within ALPHA, commitments are viewed as the mental equivalent of a contract. As such, they define a course of action/activity that the agent has agreed to, when it must realise that activity, to whom the commitment was made, and finally, what conditions, if any, would lead to it not having to fulfil the commitment. Commitment management is then a meta-level process that ALPHA agents employ to manipulate their commitments based upon some underlying strategy known as a *commitment management strategy*. This strategy specifies a set of sub-strategies that define how an agent adopts new commitments; maintains its existing commitments; refines commitments to plans into additional commitments; realises commitments to primitive actions; and handles failed commitments.

The principal sub-strategy that underpins the behaviour of ALPHA agents is commitment adoption. Commitments are adopted either as a result of a decision to realise some activity, or through the refinement of an existing activity. The former type of commitment is known as a *primary commitment* and the latter as a *secondary commitment*. The adoption of a primary commitment occurs as a result of one of two processes: (1) in response to a decision to attempt to achieve a goal using a plan of action, or (2) as a result of the triggering of a *commitment rule*. Commitment rules define situations (a conjunction of positive and negative belief atoms) in which the agent should adopt a primary commitment.

A key feature of ALPHA, which differentiates it from other agent programming languages, is the inclusion within the language of a set of programming constructs that allow the developer to explicitly specify how each agent can interact with its environment. Specifically, ALPHA includes a PERCEPTOR and an ACTUATOR construct, which specify how the agent senses and effects its environment respectively. These constructs associate Java classes that implement the sensors and effectors of an agent with the behaviour of that agent which is specified in ALPHA. The set of actuators and perceptors that are specified for a given agent is known as the *embodiment configuration* of that agent.

Engendering Reuse in ALPHA

As is described in the previous section, an ALPHA agent program traditionally takes the form of a set of commitment rules together with an initial mental state and an embodiment configuration. The ability to compose new ALPHA agent programs from pre-existing programs that are stored in different physical files, known as *role files*, was previously supported via the USE_ROLE construct. The initial motivation for the inclusion of this construct was to support the decomposition of ALPHA agent programs into their constituent roles, facilitating the reuse of those roles at compile time. However, this approach, whilst flexible, has proven to be inadequate for a number of reasons:

- The concept of a role only exists up to compile time; hence the agent is not aware, at run-time, of the role(s) that it is playing.
- The relationship that exists between the different roles is not clear - it can be viewed as either a weak form of inheritance or as composition depending on the nature of the underlying code.
- Lack of support for the templatisation of the roles makes the specification generic role implementations more difficult.

Perhaps the main cause underlying the inadequacy of this approach is that the USE_ROLE construct is, in essence, the equivalent of the \#include construct of C. Such a construct is insufficient to provide support for the composition, extension, and templatisation of roles. As a result, the construct has since been re-cast as an IMPORT construct, and APLHA has now been re-engineered to provide explicit support for roles. More importantly, by

applying the concept of a role as a run-time construct within the language, it has been possible to develop more extensive reuse mechanisms that were not available in the previous version of the language.

Role Templates

The primary construct for defining behaviours in ALPHA is the commitment rule. Informally, these rules define situations in which the agent should adopt a primary commitment (see section 2) to some activity. Traditionally, these rules were located in the body of an agent program. However, in our new framework, behaviours are defined via roles.

To facilitate the introduction of roles, we have defined a *ROLE construct*. This construct combines a unique *role identifier*, a set of commitment rules that define the behaviour that underpins the role and a set *trigger conditions* that cause the activation of the role. The identifier provides a unique way of referring to a role, and takes the form of a first-order structure whose arguments may be variables; commitments rules take the form as before, with the exception that their scope is now restricted to the role in which they are defined; and finally, the trigger conditions outline situations in which the role should be activated. Allowing the identifier to take variable arguments is the mechanism by which the role is templatised.

The instantiation of a role template is achieved through the generation of a set of variable bindings that map the arguments of the identifier to constants. This may occur in one of two ways: (1) via the satisfaction of a trigger condition, or (2) via the `activate(?role)` action. In the former case, the variable bindings are generated from the trigger condition (that is, each argument of the identifier must occur within each trigger condition). Conversely, in the latter case, the relevant variables must occur within the action definition.

We illustrate the ROLE construct through an example that defines a Subscriber facilitator role:

```
ROLE Subscriber(?name, ?topic) {
  TRIGGER BELIEF(fipaMessage(request,
              sender(?name, ?addr),
              subscribe(?topic)));

  BELIEF(fipaMessage(inform, ?sender,
           newInfo(?topic, ?content))) =>
  COMMIT(?self, ?now, BELIEF(true),
    inform(?name, newInfo(?topic, ?content)));

  BELIEF(fipaMessage(inform,
          sender(?name, ?addr),
          cancelSubscription(?topic))) =>
  COMMIT(?self, ?now, BELIEF(true),
    deactivate(Subscriber(?name, ?topic)));
}
```

In the example above, we define a Subscriber role. This role is used by middle agents, where a subscription agent subscribes to the Subscriber agent, asking to be informed whenever the Subscriber is informed of `?item` (for example, the `?item` could be status information of the form `?status`). The role is triggered whenever an agent sends a request to subscribe for information on some specified item. Whenever the Subscriber is informed of that item, it relays the item on to the subscribed agent. A second commitment rule handles the scenario in which a subscribed agent wished to stop being informed about that item.

The activation of this role can occur either as the result of a message from another agent or via the `activate(…)`action. For example, an agent that had enacted the Subscriber role was to perform the action `activate(Subscriber(Rem, fuelLevel(?level))`, then the Subscriber role would be instantiated and activated using the following variable binding {`?agent/Rem`, `?item/fuelLevel(?level)` }. This would result in all occurrences of the variable `?agent` in the commitment rules associated with the role being replaced by `Rem` and all the occurrences of `?item` being replaced by `?fuelLevel(?level)`. Also, the instance will be assigned the identifier `Subscriber(Rem, fuelLevel(?level))`. This differentiates role templates from roles and enforces the condition that each role instance must have a unique identifier.

Inheritance of Roles

Inheritance refers to the technique of extending/ polymorphing the behaviour/properties of an existing class. The value of inheritance arises from the ability to detect a set of common behaviours/properties that are shared by two or more classes, to extract them into a separate class, and to reuse the common definition in the original classes. In the context of AOP, the provision of such a technique would allow the developer to identify common behaviours and to reuse those behaviours, to extract them into an abstract role, and then to reuse that abstract role in the definition of a number of concrete roles. This has obvious benefits for developers, as it allows them to construct hierarchies of abstract roles that specify common behaviours, and then reuse those abstract roles in the design of concrete roles.

ALPHA behaviours are specified as sets of commitment rules that define situations in which the agent should act. Further, the inclusion of role templates allows the developer to specify sets of commitment rules that should only be applied by an agent under specified trigger conditions. In this context, the primary purpose of an inheritance mechanism is to allow the developer to extend the behaviour of the agent through the inclusion of additional commitment rules and through the definition of additional trigger conditions. Additionally, the developer may include additional variables in the identifier of the sub-role. Whenever an instance of the sub-role is activated, the variable binding is applied to both the sub-role and all parent roles.

Use of the extension mechanism is realised through the optional EXTENDS keyword as is shown in the example

below which illustrates how a Senior Lecturer role can be defined in terms of a Lecturer role:

```
ROLE SeniorLecturer(?subjects, ?admin)
                EXTENDS Lecturer(?subjects) {
   ... role body defined here ...
}

ROLE Lecturer(?subjects) {
  .. role body defined here ...
}
```

In this example, the SeniorLecturer role requires two parameters - the subjects that must be taught and the administrative duties that must be undertaken. Conversely, the Lecturer role only requires one parameter - the subjects that must be taught.

Composition and Aggregation of Roles

In OOP, composition and aggregation are viewed as similar techniques for building a composite object out of a number of other objects. The primary difference between these techniques arises from the lifetime of the component objects. With composition, the component objects cannot exist without the composite object. That is, if object A is composed from object B and C, then object A must be created before objects B and C, while objects B and C must be destroyed before A can be destroyed. Conversely, with aggregation, the component objects can exist before the aggregated object is created.

In the context of roles, the ability to build a role out of a set of component roles would seem to be of value. For example, consider an estate agent - the estate agent must, at different times play the role of valuer, auctioneer, and possibly salesman (in the context of showing a house to a potential bidder). However, in terms of the lifetime of the component roles, aggregation would not seem to make sense. If aggregation of roles were to be supported, then it would imply that an agent can enter into a role and, at a latter date, subsume that role into a composite role. For example, an agent should not be able to activate a salesman role and then use that instance of the role within an estate agent role. Instead, a more practical model for composite roles is that a component role is only activated after the composite role is activated. That is, an agent sells a property be activating an instance of a salesman role only after they have entered into the estate agent role.

In ALPHA, the composition of roles is supported through the inclusion of a USES construct. This construct is used within the body of a role to specify any component roles that are used by that role. To illustrate this construct, consider the estate agent role example discussed earlier in this section. In ALPHA, this role would be represented as follows:

```
ROLE EstateAgent(?area) {
  USES Valuer, Auctioneer, Salesman;
  ... role body defined here ...
}
```

This code specifies that that an Estate Agent role uses three component roles: a Valuer role, an Auctioneer role, and a Salesman role. The main purpose of the construct is to ensure, at run-time, that the component roles required to realise a composite role have been included in the compiled agent program. Also, a similar check is carried out at run-time before the activation of each role. Should one of the component roles not be defined, the instantiation of the composite role will fail and a belief outlining both the failure and the offending role will be generated.

Example: Vickrey-type Auction

To illustrate the role-based extension to ALPHA that is presented in this paper, we conclude with a simple case study of a multi-agent auction. We present a simplified auction protocol that implements a Vickrey-type auction (Vickrey 1961) that consists of a single round of bidding where the bidding agents are not aware of what each other has bid.

An overview of this protocol, modeled using an Agent UML Sequence Diagram (Bauer, Muller and Odell 2001), presented in figure 1. In this diagram, the Auctioneer agent initiates the auction by sending out a request for the Bidder agents make a bid for the specified item. The Auctioneer then waits for each of the Bidders to send back a bid in response. The Auctioneer evaluates each bid in turn, and after all the bids have been received, informs each Bidder whether or not they have won the auction.

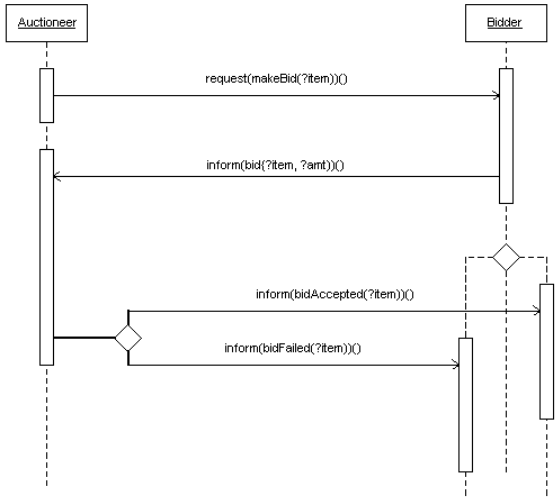

Figure 1: AUML Sequence Diagram depicting the interactions of a Vickrey-type auction

Figure 2 presents an ALPHA program that implements the Auctioneer and Bidder roles that are specified in the above auction protocol. As can be seen in this figure, the Auctioneer role is triggered by a belief that it wants to auction an item to a list of bidders, and the Bidder role is triggered by a belief that it has received a request to bid in an auction.

```
IMPORT com.agentfactory.core.fipa.agent.Agent;

LOAD_MODULE auctions module.AuctionModule;
ACTUATOR actuator.StartAuctionActuator;  //startAuction(?item, ?bidder)
ACTUATOR actuator.EndAuctionActuator;    //endAuction(?item)
ACTUATOR actuator.AddBidActuator;        //addBid(?item, ?bidder, ?amt)
PERCEPTOR perceptor.AuctionPerceptor;

ROLE Bidder(?auct, ?item) {
    TRIGGER BELIEF(message(request, sender(?auct, ?addr), makeBid(?item)));

    BELIEF(message(request, sender(?auct, ?addr), makeBid(?item))) =>
    COMMIT(?self, ?now, BELIEF(true),
      PAR(generateBid(?item),
          SEQ(AWAIT(BELIEF(bid(?amt, ?item))),
              inform(?auct, bid(?amt, ?item)),
              OR(DO_WHEN(BELIEF(message(inform, sender(?auct, ?addr), bidAccepted(?item))),
                         adoptBelief(ALWAYS(BELIEF(owner(?item))))),
                 DO_WHEN(BELIEF(message(inform, sender(?auct, ?addr), bidFailed(?item))),
                         adoptBelief(BELIEF(bidFailed(?item))))),
              deactivateRole(bidder(?auct, ?item)))));
}

ROLE Auctioneer(?item, ?bidders) {
    TRIGGER BELIEF(wantToAuction(?item, ?bidders));

    !BELIEF(auctioning(?item)) =>
    COMMIT(Self, Now, BELIEF(true), startAuction(?item, ?bidders));

    BELIEF(bidder(?bidder, ?item)) =>
    COMMIT(Self, Now, BELIEF(true),
      PAR(request(?bidder, makeBid(?item)),
          DO_WHEN(BELIEF(fipaMessage(inform,sender(?bidder, ?addr),bid(?amt, ?item))),
                  PAR(addBid(?item, ?bidder, ?amt),
                      OR(DO_WHEN(BELIEF(status(?item, ?winner, winner)),
                                 inform(?winner, bidAccepted(?item))),
                         DO_WHEN(BELIEF(status(?item, ?winner, winner)),
                                 inform(?winner, bidFailed(?item))))))));

    BELIEF(auctioning(?item)) & BELIEF(status(?item, ?winner, winner)) =>
    COMMIT(Self, Now, BELIEF(true),
      PAR(endAuction(?item),
          deactivateRole(auctioneer(?item, ?bidders))));
}
```

Figure 2: "auction.alpha" – the ALPHA Source code for a Vickrey-type auction.

As can be seen in the embodiment configuration declaration at the top of the file, actuators are included for the `startAuction()`, `endAuction()`, and `addBid()` actions. However, no actuator is provided for the `generateBid()` action. The rationale for this is simple: by making the declaration of the actuators available to an agent explicit, it is possible to specify different actuators for the same action. That is, the above code is partial code, and should not be executed directed. Instead, the developer is required to create a new ALPHA program, to import the above file, and then to add an actuator definition for the missing `generateBid()` action.

For example, to create an agent that generates a random bid in the range 0 to 1000, all that the developer must perform the following steps: (1) write an actuator unit (which is basically a Java class) whose action identifier is specified as `generateBid(?item)`, and (2) create a simple ALPHA program that imports the "auction.alpha" file and specifies the actuator created in step (1). The resulting program would look something like the program specified in figure 3.

```
IMPORT auction.alpha;

ACTUATOR actuator.MyGenerateBidActuator;
                                  //generateBid(?item)
```

Figure 3: "random.alpha" program using the example ALPHA auction program

To deploy this agent, the developer simply compiles the source code into an ALPHA deployment file (denoted by a ".agent" extension), writes the necessary agent platform script (Collier 2005), and then starts an agent platform. The screenshots in figure 4 shows a view of an Agent Factory agent platform that contains three agents: Rem, Fred, and Joe. The screenshots show Joe's commitment

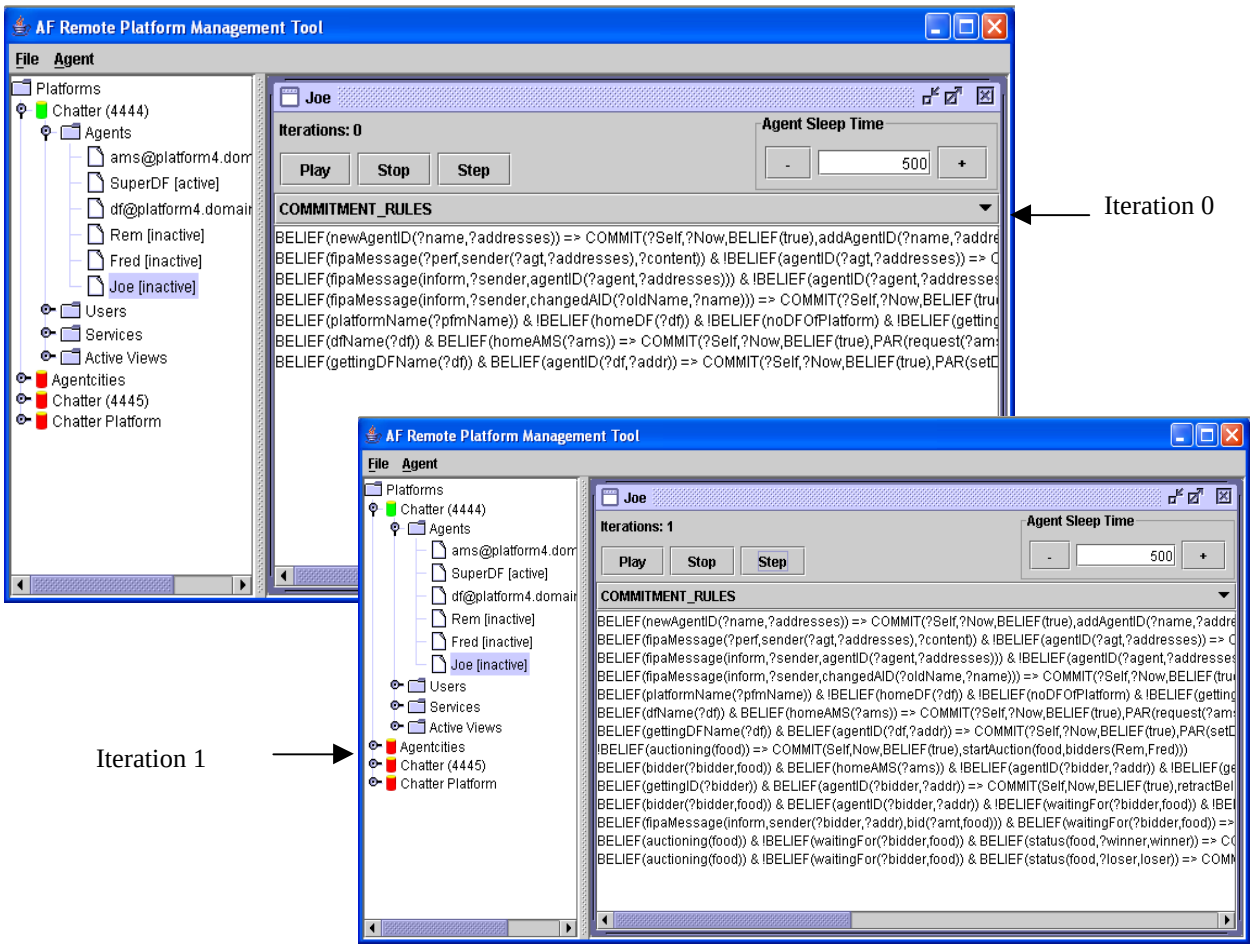

Figure 4: Commitment Rules of Joe after iterations zero and one

rules after the first and second iterations. Joe is given the belief:

```
BELIEF(wantToAuction(food, bidders(Rem, Fred)))
```

As can be seen in the screenshots, this belief causes Joe to trigger an instance of the Auction role. The first screenshot at iteration zero shows that Joe has no commitment rules relating to this role, while the screenshot at iteration one shows that Joe has commitment rules relating to this role. Once Joe creates an instance of the Auction role, he sends a request for bids for some food to Rem and Fred.

Eventually, this causes Rem and Fred to trigger an instance of the Bidder role, which requires that they each make a random bid. These agents then inform Joe of their respective bids. This is illustrated through the screenshot of Joe's beliefs (see figure 5), where it can be seen that Joe believes that he has received bids of 440 and 241 from Rem and Fred respectively. Joe awards the food to the Bidder the made the largest bid, which in this case is Rem. Those agents that lost the bid are informed of their failure, and each of the participating agents deactivates the corresponding role instance.

This results in the role instance being destroyed and the corresponding commitment rules being removed from the agents' commitment rule set.

Discussion

The concept of a role is becoming increasingly important in the modelling of multi-agent systems. Recent work in the area of AOP languages, and in particular, 3APL (Dastani et al. 2004), has begun to focus on how the concept of a role might be applied in this area. Specifically, the authors present a formal model of a role-playing agent and specify additional constructs that support the activation and deactivation of roles.

In this paper, we present an extension to the ALPHA agent programming language that, while providing similar support for the activation and deactivation of roles (here role activation is also known as *role instantiation*), focuses on how the concept of a role can be used to engender support for reuse of agent programs. Specifically, we introduce the concept of a role template, and show how OOP reuse mechanisms such as inheritance and composition may be employed within an AOP setting.

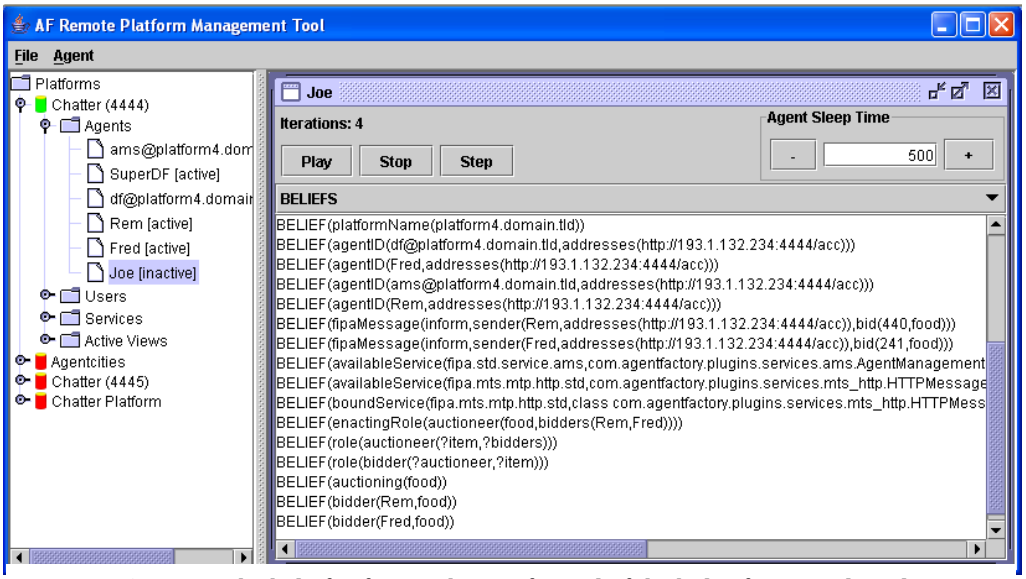

Figure 5: The beliefs of Joe as he is informed of the bids of Rem and Fred

We view this work as a vital first step in our investigation of how the concept of a role can be used in the context of AOP to support the implementation of multi-agent systems that are able to adapt their behaviour at run-time. Specifically, future work in this area will investigate: (1) an infrastructure to support the location and enactment of roles at run-time, (2) the design of introspective capabilities that will allow agents to reason about their roles, and (3) the extension of our role concept to support social programming constructs.

References

Bauer, B., Muller, J. P., and Odell, J., 2001. *Agent uml: A formalism for specifying multiagent interaction*. In Agent-Oriented Software Engineering (eds. P. Ciancarini and M. Wooldridge),, Springer Verlag.

Collier, R., 2001: Agent Factory: A Framework for the Engineering of Agent-Oriented Applications, Ph.D. Thesis, Department of Computer Science, University College Dublin.

Collier, R., O'Hare, G. M. P., Lowen, T. D., and Rooney, C. F. B., 2003: *Beyond Prototyping in the Factory of Agents*, in Proc. 3rd Int. Central and Eastern European Conference on Multi-Agent Systems (CEEMAS), Prague, Czech Republic, Springer Verlag.

Dastani, M., van Riensdijk, B., Dignum, F., and Meyer, J-J Ch., 2003: *A programming language for cognitive agents: Goal directed 3apl*, in Proc. of AAMAS2003, Melbourne.

Dastani, M., Birna van Riems-dijk, M., Hulstijn, J., Dignum, F., and Meyer J-J. Ch., 2004: *Enacting and deacting roles in agent programming*, in Proceedings of the 2nd International Workshop on Programming Multi-Agent Systems (PROMAS2004), Springer Verlag.

Dickinson, I. and Wooldridge, M., 2003: *Towards practical reasoning agents for the semantic web*, in 2nd Int. Joint Conf. on Autonomous Agents and Multi-Agent Systems (AAMAS-03), Melbourne, Australia.

Karageorgos, A., Thompson, S. and Mehandjiev, N., 2003: *Specifying Reuse Concerns in Agent System Design Using a Role Algebra.* In: Agent Technologies, Infrastructures, Tools, and Applications for e-Services. Lecture Notes in Artificial Intelligence LNAI, 2592. Springer-Verlag.

Muldoon, C., O'Hare, G.M.P., Phelan, D., Strahan, R., and Collier, R., 2003: *ACCESS: An Agent Architecture for Ubiquitous Service Delivery*, Proc 7th Int'l Workshop on Cooperative Information Agents (CIA2003), Helsinki.

Nwana, H., Ndumu, D., Lee, L., and Collis, J., 1999: *Zeus: A toolkit for building distributed multi-agent systems*. Applied Artificial Intelligence Journal, 13(1):129–186.

Odell, J., Van Dyke Parunak, H., and Bauer, B., 2001: *Representing agent interaction protocols in UML*. In Paolo Ciancarini and Michael Wooldridge, editors, Agent-Oriented Software Engineering, Springer-Verlag.

Odell, J., Van Dyke Parunack, H., Brueckner, S., and Sauter J., 2003: *Temporal aspects of dynamic role assignment*, in Proceedings of the 4th International Workshop on Agent-Oriented Software Engineering (AOSE2003).

O'Hare, G. M. P., and O'Grady, M. J., 2003: *Gulliver's Genie: A Multi-Agent System for Ubiquitous and*

Intelligent Content Delivery, In Press, Computer Communications, Elsevier Press.

Rao, A., 1996: *AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language*, In de Velde, W., Perram, W.J.V., eds.: Proceeding of the 7th International Workshop on Modeling Autonomous Agents in a Multi-Agent World, Eindhoven, The Netherlands.

Ross, R., Collier, R., and O'Hare, G. M. P., 2004: *AF-APL: Bridging principles & practices in agent oriented languages*, in Proc. 2nd International Workshop on Programming Multiagent Systems Languages and tools (PROMAS2004), New York, USA.

Ubayashi, N., and Tamai, T., 2000: *RoleEP: role based evolutionary programming for cooperative mobile agent applications*, in the International Symposium on Principles of Software Evolution, Kanazawa, Japan.

Vickrey, W., 1961: *Counter speculation, auctions, and competitive sealed tenders*, Journal of Finance, 16(1):8–37.

Shoham, Y., 1993: *Agent-Oriented Programming*, Artificial Intelligence (60), pp 51-90

Collier, R., 2005: *The Agent Factory Platform Administrators Guide*, url: http://www.agentfactory.com/ agentfactory/manual