Realising Reusable Agent Behaviours with ALPHA

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

[1]Practice & Research in Intelligent Systems & Media (PRISM) Laboratory, Department of Computer Science, University College Dublin (UCD), Belfield, Dublin 4, Ireland.
{rem.collier, gregory.ohare}@ucd.ie
[2]Department of Computer Science, University of Bremen, Germany.
robertr@tzi.de

Abstract. This paper describes a revision to the design of Agent-Oriented Programming (AOP) that introduces the concept of a role. The proposed AOP framework introduces the notion of a role template and describes how these templates can be used to engender code reuse. We then use this framework to extend the ALPHA programming language and illustrate the use of this extension via a simple case study.

1 Introduction

To date, support for the reuse of program code has not been an issue in the design of Agent-Oriented Programming (AOP) languages. This is in stark contrast with other programming paradigms, such as Object-Oriented Programming and Component-Based Programming, where reuse is a core issue. It is also in contrast with other approaches to agent development. One example is Jade [2], which employs Java's built-in reuse mechanisms to support reusable behaviours. Support for code reuse has also been implemented for the Zeus tool [11]. In fact, the authors point out in this paper that the lack of support for reuse was a significant problem for the commercial developers who used earlier versions of Zeus [13].

The failure of AOP researchers to consider the issue of reuse reflects the current state of the research area, which has tended to focus answering the question: *what are the most appropriate features for an agent programming language?* However, at this point in time, there are a number of more established agent programming languages, such as 3APL [6], AgentSpeak(L) [18] and Nuin [9], which represent viable alternatives for the implementation of agent-oriented applications. To properly evaluate the potential of languages such as these, it is vital that the issue of reuse be addressed. Only then, will it be possible to evaluate their potential through their use in large projects that attempt to solve real world problems.

This paper presents details of how reuse has been engendered in the ALPHA programming language [19], which sits at the heart of the Agent Factory framework [4] [5]. 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 [4], in the development of a number of real world application domains [16] [12].

2 Reuse and Agent Factory

Reuse has long been a central issue in the design of Agent Factory (AF) [4] [5]. Since its inception, AF has included some form of AOP language that provided support for the programming of intentional agents that reasoned about how best to act through some form of mental state architecture. Both Agent Factory, and the underlying programming language have undergone a series of evolutions that have brought it from its initial form, as described in [4] to its present form, as described in [19].

While details of these earlier versions of AF are not relevant to this paper, one aspect that is relevant is the reuse mechanisms that underpinned them. In the first versions of this language, reuse was engendered through a simple inheritance mechanism [2]. Conceptually, an agent program was decomposed into a set of *agent classes* that were organized into a hierarchy, the root of which was the **Agent** class. This base class contained a partial agent program that was common to all agents. Whenever a developer wished to create a new type of agent, they were required to extend an existing agent class, and add any addition agent code. When the developer wished to create an agent, they simply selected the agent class that they wished to instantiate, and an underlying instantiation mechanism constructed the agent program and instantiated the agent.

A key drawback of this approach was the lack of flexibility that ensued from the use of simple inheritance. As the complexity of the applications to which AF was being applied increased, an increasing number of scenarios arose where composition would have been more appropriate that inheritance. Recognition of this limitation drove work on a more flexible composition-based reuse mechanism for AF that was based on the concept of *roles* [17] as implementation-time constructs. Some details of this model are presented in section 5

3 Reusing Code through Roles

Roles are widely recognized as a key concept in the analysis and design of multi-agent systems [8, 14, 22] and are often viewed as a valuable level of abstraction, which can be used to define common behaviours that may be reused in the design of many different types of agent. Based on this, some researchers have argued that there is a need for a new class of *Role-Oriented Programming Environment* [15]. However, a recent survey of role-based approaches for agent development [3], found that the notion of a role is less well established with respect to the implementation of agents.

One exception to this is the approach presented in [11], 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* [19], 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.

In the context of AOP, [7] presents an extension to the pre-existing 3APL programming language [6] to include 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 how this model might be used to engender reuse within the language.

The work presented in this paper builds on the extension proposed in [7] by considering how the concept of a role might be used to engender reuse within AOP languages. Specifically, we explore how OOP reuse mechanisms such as inheritance, composition, and aggregation can be applied to AOP. However, a key difference between the approach presented here, and that presented in [7] 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 [14], 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 happens regularly 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. [13] 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. This template is then instantiated for each situation in which the common protocol is required.

In summary, this paper proposes an extension to AOP in which an agent is modelled as a set of role templates that can be instantiated as appropriate. We then introduce a number of OOP-inspired reuse mechanisms that can be used by developers when constructing agent-oriented applications using an AOP language.

3.1 Composition and Aggregation of Roles

From an OOP perspective, composition and aggregation are 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.

3.2 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 would allow them to construct hierarchies of abstract roles that specify common behaviours, and then reuse those abstract roles in the design of concrete roles. An example of this approach was employed an earlier version of the ALPHA programming language, known as AF-APL, where a set of behaviours (e.g. address book management) that are common to all 'socially able' FIPA-ACL [10] speaking agents was reused by every developer [4].

4 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 [19], and for a more detailed overview of the formal model that underpins the syntax and semantics of an earlier version of this language, known as AF-APL, the reader is directed to [4].

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* [4].

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.

5 Supporting Roles in ALPHA

As s 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 discussing in section 3. 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. This support is realised through the inclusion of a ROLE construct that has been designed in line with the revised AOP framework discussed in section 3.

5.1 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 4) 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 (section 3).

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(?agent, ?item) {
  TRIGGER BELIEF(fipaMessage(request, sender(?name, ?addr), subscribe(?item)));

  BELIEF(fipaMessage(inform, ?sender, ?item)) =>
  COMMIT(Self, Now, BELIEF(true), inform(?agent, ?item));

  BELIEF(fipaMessage(inform, sender(?agt, ?addr), cancelSubscription(?item))) =>
  COMMIT(Self, Now, BELIEF(true), deactivate(Subscriber(?agt, ?item)));
}
```

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.

5.2 Inheritance of Roles

The ALPHA inheritance mechanism (see section 3.2) supports only the extension of a role - it does not allow polymorphism. When an existing role is extended, the developer may specify additional commitment rules and trigger conditions in the sub-role. 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.

5.3 Composition of Roles

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. For example, section 3.1 used an estate agent role to illustrate the applicability of composition with respect to roles. The segment of code below illustrates how this role can be represented in ALPHA:

```
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.

6 Example: Vickrey-type Auction

To illustrate the framework 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 [21] that consists of a single round of bidding where the bidding agents are not aware of what each other has bid.

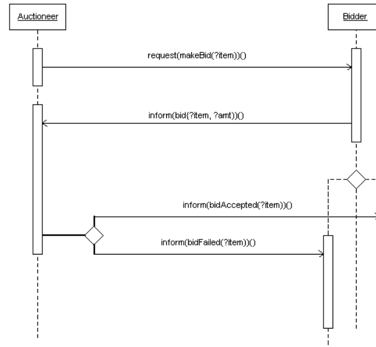

Fig. 1: AUML Sequence Diagram depicting the interactions of a Vickrey-type auction

An overview of this protocol, modelled using an Agent UML Sequence Diagram [1], 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 2 above 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.

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 implemented 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. This is illustrated below:

```
IMPORT auction.alpha;

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

BELIEF(wantToAuction(food, bidders(rem, bob)));
```

```
  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(fipaMessage(request, sender(?auct, ?addr), makeBid(?item)));
```

```
    BELIEF(fipaMessage(request, sender(?auct, ?addr), makeBid(?item))) =>
    COMMIT(Self, Now, BELIEF(true), generateBid(?item));

    BELIEF(bid(?amt, ?item)) =>
    COMMIT(Self, Now, BELIEF(true), PAR(inform(?auct, bid(?amt, ?item)),
                               adoptBelief(ALWAYS(BELIEF(bidded(?amt, ?item))))));

    BELIEF(fipaMessage(inform, sender(?auct, ?addr), bidAccepted(?item))) &
    BELIEF(bidded(?amt, ?item)) =>
    COMMIT(Self, Now, BELIEF(true),
        PAR(retractBelief(ALWAYS(BELIEF(bidded(?amt, ?item)))),
            adoptBelief(ALWAYS(BELIEF(owner(?item)))),
            deactRole(bidder(?auct, ?item))));

    BELIEF(fipaMessage(inform, sender(?auct, ?addr), bidFailed(?item))) &
    BELIEF(bidded(?amt, ?item)) =>
    COMMIT(Self, Now, BELIEF(true),
        PAR(retractBelief(ALWAYS(BELIEF(bidded(?amt, ?item)))),
            deactRole(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)) & !BELIEF(waitingFor(?bidder, ?item)) &
    !BELIEF(status(?item, ?bidder, ?status)) =>
    COMMIT(Self, Now, BELIEF(true), PAR(request(?bidder, makeBid(?item)),
                  adoptBelief(ALWAYS(BELIEF(waitingFor(?bidder, ?item))))));

    BELIEF(fipaMessage(inform, sender(?bidder, ?addr), bid(?amt, ?item))) &
    BELIEF(waitingFor(?bidder, ?item)) =>
    COMMIT(Self, Now, BELIEF(true), PAR(addBid(?item, ?bidder, ?amt),
                  retractBelief(ALWAYS(BELIEF(waitingFor(?bidder, ?item))))));

    BELIEF(auctioning(?item)) & !BELIEF(waitingFor(?bidder, ?item)) &
    BELIEF(status(?item, ?winner, winner)) =>
    COMMIT(Self, Now, BELIEF(true), PAR(inform(?winner, bidAccepted(?item)),
             endAuction(?item), deactivateRole(auctioneer(?item, ?bidders))));

    BELIEF(auctioning(?item)) & !BELIEF(waitingFor(?bidder, ?item)) &
    BELIEF(status(?item, ?loser, loser)) =>
    COMMIT(Self, Now, BELIEF(true), inform(?loser, bidFailed(?item)));
  }
```

Fig. 2. “auction.alpha” – the ALPHA Source code for a Vickrey-type auction.

7 Discussion

This paper presents a framework for reuse in AOP languages that is founded on the notion of a role. Specifically, this framework presents a model of agents whose behaviour is specified as a set of roles that can be dynamically activated and deactivated at run-time. In addition, we argue that the most appropriate approach to representing the roles that an agent can play is through the use of role templates. These templates are, in essence, parameterised definitions of the expected behaviours that an agent playing that role should realise. Further, we have motivated the inclusion of two reuse mechanisms, namely composition and inheritance. These mechanisms enable developers to easily construct new roles that are based upon pre-existing roles.

This work introduces the first AOP language that implements support for reuse through a combination of role templates and OOP inspired reuse mechanisms such as inheritance and composition. It is out view that the inclusion of such support is vital if AOP languages are to become a viable option for the development of agent-oriented applications. As a result, we view the work presented in this paper to be a significant step in this direction.

References

[1] B. Bauer, J. P. Muller, and J Odell. *Agent uml: A formalism for specifying multiagent interaction*. In Paolo Ciancarini and Michael Wooldridge, editors, *Agent-Oriented Software Engineerin*g. Springer Verlag, 2001.

[2] F. Bellifemine, A. Poggi, G. Rimassa: *JADE – A FIPA-compliant agent framework*, in Proceedings of the 4th International Conference and Exhibition on The Practical Application of Intelligent Agents and Multi-Agents (PAAM), London, UK 1999.

[3] G. Cabri, L. Ferrari, L. Leonardi, F. Zambonelli, *Role-based Approaches for Agent* Development, in Proceedings of the 3rd International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS-04) , NY, 2004

[4] R. W. Collier. *Agent Factory: A Framework for the Engineering of Agent-Oriented Application*s. PhD Thesis, Dept. of Computer Science, Univ. College Dublin, 2001.

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

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

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

[8] S. A. DeLoach. *The MASE methodology*, Methodologies and Software Engineering for Agent Systems, 11, 2004.

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

[10] FIPA. *Fipa 2000 standards*. URL: http://www.fipa.org, 2000.

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

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

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

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

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

[16] G. M. P. O'Hare and M. J. O'Grady, *Gulliver's Genie: A Multi-Agent System for Ubiquitous and Intelligent Content Delivery*, In Press, Computer Communications, Elsevier Press, 2003.

[17] I. Partsakoulakis and G. Vouros, *Importance and properties of roles in MAS organization: A review of methodologies and systems*, in Proceedings of the workshop on MAS Problem Spaces and Their Implications to Achieving Globally Coherent Behavior, Bologna, Italy, 2002.

[18] A. Rao, *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, 1996.

[19] R. Ross, R. Collier, and G. O'Hare. *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, 2004.

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

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

[22] M. Wooldridge, N. R. Jennings, and D. Kinny, *The gaia methodology for agent-oriented analysis and design*, Autonomous Agents and Multi-Agent Systems, 3(3):285–312, 2000.