# Specifying Reuse Concerns in Agent System Design Using A Role Algebra

Anthony Karageorgos
PhD Student
Dept. of Computation
UMIST, Manchester M60 1QD, UK
+44-161-2003306
karageorgos@computer.org

Simon Thompson
Team Leader
Intelligent Agents Research Group, BT Exact Technologies
+44-1473 605531
simon.2.thompson@bt.com

Nikolay Mehandjiev
Lecturer
Dept. of Computation
UMIST, Manchester M60 1QD, UK
+44-161-2003319
nikolay@computer.org

During the design of an agent system many decisions will be taken that determine the structure of the system, for reasons that are clear to the designer and customers at the time. However, when later teams approach the system it may not be obvious why particular decisions have been taken. This problem is particularly acute in the case of designers attempting to integrate services from many different service providers. In this paper a mechanism for recording these decisions is described. We illustrate how design decisions can be captured and how these abstractions can then be used in as the basis of reuse in an extension of the Zeus agent toolkit.

## Introduction

Multi-agent system architectures can be naturally viewed as organised societies of individual computational entities e.g. [6, 15, 19], and hence the problem of designing a multi-agent system refers to designing an *agent organisation* The criteria affecting an agent organisation design decision are numerous and highly dependent on factors that may change dynamically. Therefore, there is no standard best organisation for all circumstances [14, 15]. As a result, agent organisation design rules are left vague and informal, and their application is mainly based on the creativity and the intuition of the human designer. This can be a serious drawback when designing large and complex real-world agent systems. Therefore, many authors argue that social and organisational

abstractions should be considered as first class design constructs and that the agent system designer should reason at a high abstraction level, e.g. [9, 12].

## Designing agent organisations

Early research prototypes of agent-based systems were built in an ad-hoc manner. However, the need to engineer agent systems solving real-world problems has given rise to a number of systematic methodologies for agent oriented analysis and design such as MESSAGE [5], GAIA [19] and SODA [12]. All these methodologies involve a number of analysis and design sub-models emphasising particular analysis and design aspects.

Existing approaches to designing agent systems could be further improved in the following ways:

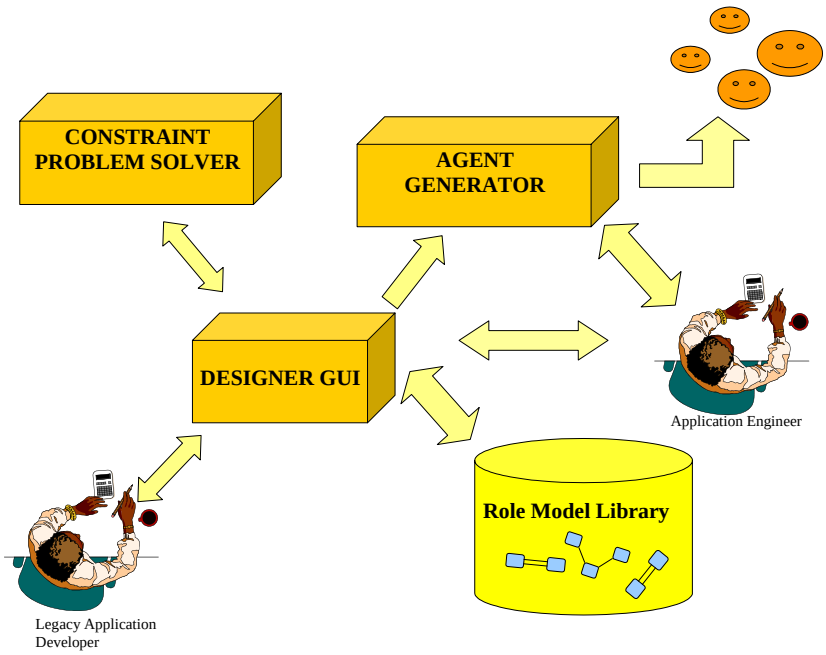


**Figure 1. Conceptual View of Proposed System**

- A more systematic way to construct large agent system design models from the analysis models. The steps involved in transforming analysis to design models are not specified to a detail that enables at least some degree of automation by a software tool

[16].

- By considering non-functional requirements on design time. The aim should be to achieve as optimum organisation on design time as possible. To achieve this, some means for considering non-functional requirements before actually deploying the multi-agent system is needed. This hypothesis is along the lines of similar works [13, 14] where the behaviour of a multi-agent system is modelled and studied before actual system deployment.
- By reusing organisational settings. This view regarding reuse of organisational settings has been inspired by the concepts introduced in [21]. It is believed that that work can be further extended by classifying known organisational patterns, and by providing some rigorous means for selecting them in a particular design context.

In this paper we examine how these concerns can be addressed and how we can implement an environment that supports the reuse of design knowledge using the abstractions of role modelling that we have developed.

## Our Approach

Figure 1 shows an overview of our approach. Our objective is to develop a technology that supports engineers in describing the design concerns that have motivated choices about models of implementation in particular systems, and to place this knowledge in a repository. The repository would then be used by subsequent engineers who wished to reuse subsystems or to modify and rebuild the legacy system itself.
We believe that further value can be added to the system by providing advice to the engineer using constraint satisfaction technology that can provide possible solutions to compositional design problems.
Finally the resulting designs can be used to generate template systems linked to libraries of domain specific implementation code.
The rest of this paper can be divided into three parts. We will describe the role modeling abstractions that we have developed; we then show how they can provide adequate representational power to capture some of the design knowledge in an example derived from a real development project. Finally we describe how we have implemented a prototype that demonstrates that this knowledge can be used to support design reuse.

**Background**

Existing role-based approaches to multi-agent system design stress the need to identify and characterise relations between roles [1, 9]. However, only a small number of approaches investigate the consequences of role relations on the design of multi-agent

systems, e.g. [9]. This is partly due to lack of formal foundations of role relationships. Therefore, in this work we first identified role relations that would affect multi-agent system design and subsequently we formalised them in an algebraic specification model. Role identification was based on organisational principles and in particular on *role theory* [3].

Role theory emphasises that various relations may exist between roles. For example, an examiner cannot also be a candidate at the same time and therefore appointing these roles to a person at the same time results to inconsistency. Role relations can be complex. For example, a university staff member who is also a private consultant may have conflicting interests. In this case, appointing these roles to the same person is possible but it would require appropriate mechanisms to resolve the conflicting behaviour.

**Role characteristics**

Following [9], a role is defined as associated with a *position* and a set of *characteristics*. Each characteristic includes a set of *attributes*. Countable attributes may further take a range of values. More specifically, a role is capable of carrying out certain *tasks* and can have various *responsibilities* or *goals* that aims to achieve. Roles normally need to interact with other roles, which are their *collaborators*. Interaction takes place by exchanging messages according to *interaction protocols*. A collection of interacting roles representing collective behaviour constitutes a *role model.*

Roles can be extended to create specialised roles by a process called role *specialisation* or *refinement* [1, 9]. Specialised roles represent additional behaviour on top of the original role behaviour in a manner similar to inheritance in object-oriented systems.

In order for roles to pragmatically represent behaviour in an application domain, they need to model issues relevant to non-functional requirements in that domain. Therefore, the above role definition is extended to include *performance variables*. Performance variables are parameters whose value defines the run-time behaviour represented by a role. For example, if the behaviour a role represents requires using some resource like storage space, the resource capacity can be modelled by a performance variable. Performance variables can also be defined at an agent level. In that case, their value is a function of the function of the respective performance variables of all roles the agent is capable of playing. This allows us to apply design heuristics by imposing constraints on the values of the agent performance variables that must be observed when allocating roles to agents. This is illustrated in the example given below

## A role algebra for agent system design

Based on role theory [3] and on case studies of human activity systems, e.g. [17], six basic role relations have been identified. In this section, a formal model of role relations is defined, referred by the term *role algebra.* Using relations from the role algebra, constraints driving the assignment of roles to agents can be specified and hence the agent organisation design process can be partially automated.

### Relations of the Role Algebra

Let $R$ be a set of roles. For any $r_1, r_2 \in$ R, the following binary relationships may hold:

1) **Equals (eq)** — This means that $r_1$ and $r_2$ describe exactly the same behaviour. For example, the terms *Advisor* and *Supervisor* can be used to refer to people supervising PhD students. When two roles are equal, an agent playing the one role also plays the other at the same time. The relation *Equals* $\subseteq R \times R$ is an equivalence relation since it is reflexive, symmetric and transitive:

a) $\forall\, r : R \;\; (r \text{ eq } r)$

b) $\forall\, (r_1, r_2) : R \times R \; (r_1 \text{ eq } r_2 \Rightarrow r_2 \text{ eq } r_1)$

c) $\forall\, (r_1, r_2, r_3) : R \times R \times R \; ((r_1 \text{ eq } r_2) \wedge (r_2 \text{ eq } r_3) \Rightarrow (r_1 \text{ eq } r_3))$

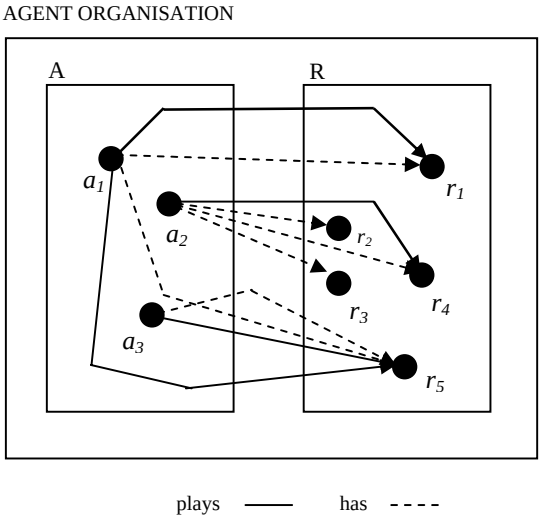


**Figure 2: Semantics of role relations**

2) **Excludes (not)** — This means that $r_1$ and $r_2$ cannot be assigned to the same agent simultaneously. For example, in a conference reviewing agent system, an agent should not be playing the roles of paper author and paper reviewer at the same time. Furthermore, a role cannot exclude itself — if it would then no agent would ever play it. Therefore, the relation *Excludes* $\subseteq R \times R$ is anti-reflexive and symmetric:

a) $\forall\, r : R \;\; (\neg(r \text{ not } r))$

b) $\forall\, (r_1, r_2) : R \times R \; (r_1 \text{ not } r_2 \Rightarrow r_2 \text{ not } r_1)$

3) **Contains (in)** — This means that a role is a sub-case/specialisation of another role. Therefore, the behaviour the first role represents completely includes the behaviour of the second role. For example, a role representing *Manager* behaviour completely contains the behaviour of the *Employee* role. When two roles such that the first contains the second are composed, the resulting role contains the characteristics of the first role only. Therefore, the relation *Contains* $\subseteq R \times R$ is reflexive and transitive:

a) $\forall\, r : R \; (r \text{ in } r)$

b) $\forall\, (r_1, r_2, r_3) : R \times R \times R \; ((r_1 \text{ in } r_2) \wedge (r_2 \text{ in } r_3) \Rightarrow (r_1 \text{ in } r_3))$

4) **Requires (and)** — The *Requires* relation can be used to describe that when an agent is assigned a particular role, then it must also be assigned some other specific role as well. This is particularly applicable in cases where agents need to conform to general rules or

play organisational roles. For example, in a university application context, in order for an agent to be a *Library_Borrower* it must be a *University_Member* as well. Although the behaviour of a *Library_Borrower* could be modelled as part of the behaviour of a *University_Member*, this would not be convenient since this behaviour could not be reused in other application domains where being a *Library_Borrower* is possible for everyone. Furthermore, each role requires itself. Intuitively, the roles that some role *r* requires are also required by all other roles that require *r*. Therefore, the relation *Requires* $\subseteq R \times R$ is reflexive, and transitive:

a) $\forall\ r : R\ (r \text{ and } r)$

b) $\forall\ (r_1, r_2, r_3) : R \times R \times R\ ((r_1 \text{ and } r_2) \wedge (r_2 \text{ and } r_3) \Rightarrow (r_1 \text{ and } r_3))$

5) **Addswith (add)** — The *Addswith* relation can be used to express that the behaviours two roles represent do not interfere in any way. For example, the *Student* and the *Football_Player* roles describe non-excluding and non-overlapping behaviours. Hence, these roles can be assigned to the same agent without any problems. The relation *Addswith* $\subseteq R \times R$ is reflexive and symmetric:

a) $\forall\ r : R\ (r \text{ add } r)$

b) $\forall\ (r_1, r_2) : R \times R\ ((r_1 \text{ add } r_2) \Rightarrow (r_2 \text{ add } r_1))$

6) **Mergeswith (merge)** — The *Mergeswith* relation can be used to express that the behaviours of two roles overlap to some extend or that different behaviour occurs when two roles are put together. For example, a *Student* can also be a *Staff_Member*. This refers to cases when PhD students start teaching before they complete their PhD or they register for another degree (e.g. an MBA) after their graduation. Although members of staff, these persons cannot access certain information (e.g. future exam papers) due to their student status. Also, their salaries are different. In cases like this, although the two roles can be assigned to the same agent, the characteristics of the composed role are not exactly the characteristics of the two individual roles put together. The relation *Mergeswith* $\subseteq R \times R$ is symmetric:

a) $\forall\ (r_1, r_2) : R \times R\ (r_1 \text{ merge } r_2 \Rightarrow r_2 \text{ merge } r_1)$

**Semantics of role relations**

To describe the semantics of role relations we represent an agent organization by a two-sorted algebra (fig. 1). The algebra includes two sorts, *A* representing agents and *R* representing roles.

Let *Has*: $A \rightarrow R$ be a relation mapping agents to roles. The term "has" means that a role has been allocated to an agent by some role allocation procedure or tool. It is possible for an agent to have roles that do not contribute to defining the agent behaviour. For example, this happens when roles merge with other roles. For each $a \in A$, let *a.has* be the set of roles that the agent *a* maps to in the relation *Has*. In other words, *a.has* denotes the relational image of the singleton $\{a\} \subseteq A$ in the relation *Has.*

Let *Plays*: $A \rightarrow R$ be a relation mapping agents to roles again. The term "plays" means that that the behaviour a role represents is actively demonstrated by the agent, for example the role does not merge with other roles that are also played by the agent. For each $a \in A$, let *a.plays* denote the set of roles that the agent *a* maps to in the relation *Plays*. In other words, *a.plays* denotes the relational image of the singleton $\{a\} \subseteq A$ to the relation *Plays*.

The meaning of the relations between roles can now be described as follows:

- **Equals** — An agent has and plays equal roles at the same time.

  $\forall\, a : A, (r_1, r_2) : R \times R \cdot (r_1 \text{ eq } r_2 \Leftrightarrow ((r_1 \in a.has \Leftrightarrow r_2 \in a.has) \wedge (r_1 \in a.plays \Leftrightarrow r_2 \in a.plays)))$

- **Excludes** — Excluded roles cannot be assigned to the same agent.

  $\forall\, a : A, (r_1, r_2) : R \times R \cdot (r_1 \text{ not } r_2 \Leftrightarrow \neg(r_1 \in a.has \wedge r_2 \in a.has))$

- **Contains** — Contained roles must be assigned and played by the same agent as their containers.

  $\forall\, a : A, (r_1, r_2) : R \times R \cdot (r_1 \text{ in } r_2 \Leftrightarrow ((r_2 \in a.has \Rightarrow r_1 \in a.has) \wedge (r_2 \in a.plays \Rightarrow r_1 \in a.plays)))$

- **Requires** — Required roles must be played by the same agent as the roles that require them.

  $\forall\, a : A, (r_1, r_2) : R \times R \cdot (r_1 \text{ and } r_2 \Leftrightarrow (r_1 \in a.plays \Rightarrow r_2 \in a.plays))$

- **MergesWith** — When two roles merge, only the unique role that results from their merge is played by an agent.

  $\forall\, a : A, (r_1, r_2) : R \times R \cdot (r_1 \text{ merge } r_2 \Leftrightarrow \exists 1\, r_3 : R \cdot((r_1 \in a.has \wedge r_2 \in a.has) \Rightarrow (r_1 \notin a.plays \wedge r_2 \notin a.plays \wedge r_3 \in a.has)))$

  For example, let us assume that roles $r_2$ and $r_3$ merge resulting to role $r_4$. Based on the above semantic definition, if an agent has $r_2$ and $r_3$ then it must also have $r_4$ and it must not play $r_2$ and $r_3$ (the agent may or may not play $r_4$ depending on the relations of $r_4$ with the other roles the agent has). The example of a Mergeswith

relation between roles $r_2$, $r_3$, and $r_4$, where $r_4$ is played by the agent, is depicted in fig. 1.

- **AddsWith** — There is no constraint in having or playing roles that add together.

  $\forall\ a : A, (r_1, r_2) : R \times R \cdot (r_1 \text{ add } r_2 \Leftrightarrow (r_1 \in a.has \Rightarrow ((r_2 \in a.has \vee r_2 \notin a.has) \wedge (r_2 \in a.plays \vee r_2 \notin a.plays))))$

Using the above semantic axioms, it is trivial to verify that the properties of role relations that we have introduced hold.

Furthermore, relations between more than two roles can be defined in a similar manner. In that case, a predicate notation is more convenient to represent role relations. For example, when three roles $r_1$, $r_2$, and $r_3$ merge to $r_4$ this can be noted by merge($r_1$, $r_2$, $r_3$, $r_4$). In this paper, we will not provide any formal definitions for relations among roles with arity greater than two.

Role relations, as defined in the above algebra, constrain the way that roles can be allocated to agents. Therefore, the agent organisation design problem is transformed to a constraint satisfaction problem that must be solved for roles to be allocated to agents. The problem can be constrained further by including constraints based on general design heuristics. These constraints are expressed on the performance variables of the agents. For example, the system designer should be able to define the maximum number of roles that an agent could play, or an upper limit to the resource capacity that the roles an agent plays would require. Furthermore, role allocation heuristics could also be specified. For example, roles requiring access to similar resources could be required to be assigned to the same agent.

## Example: supporting mobile work teams

For this example we consider a case study concerning telephone repair service teams. The aim is to build an agent system that would assist field engineers to carry out their work. Among the issues involved in such a system are those of travel management, teamwork coordination, and expertise knowledge management [17, 18].

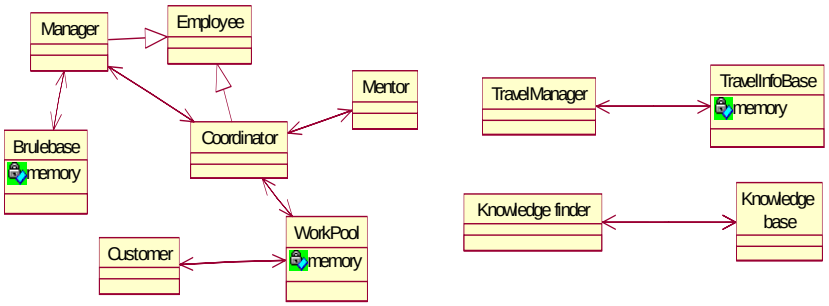


**Figure 3: Role models for the telephone repair**

Travel management is about support to mobile workers for moving from one repair task location to another. It involves finding the position of each worker, obtaining relevant travel information, planning the route to the next repair task location and allocating travel resources as required. Teamwork coordination is about allocating and coordinating the execution of repair tasks in a decentralised manner taking into account the personal preferences and working practices of the mobile workers. Work knowledge management concerns storage and dissemination of expertise work knowledge.

**Role identification**

In order to model the above system in terms of roles, the first thing to do is to identify the roles involved in the case study. According to [10] a way to identify roles in an application domain is to start from identifying use cases, associating each use case with a goal, creating a goal hierarchy from the use case hierarchy and coalescing semantically relevant goals in roles. For the purpose of our example we consider the following three use cases: *Teamwork coordination*, *Travel management* and *Work Knowledge Management.*

Applying the methodology suggested in [10], the following roles can be identified (fig. 2):

1. *Employee*: This role describes generic behaviour of the members of the customer service teams. An example of this type of behaviour is accessing common team resources including work practice announcements and business news.

2. *Coordinator:* The Coordinator role describes the behaviour required to coordinate the work of a field engineer. This includes bidding for and obtaining repair work tasks from a work pool, negotiating with other workers and the team manager as required and scheduling and rescheduling work task execution.

3. *Manager:* The Manager role models the behaviour of the team manager. This includes confirming task allocation, monitoring work and ensuring that business rules are followed.

4. *Mentor:* The mentor role provides assistance to field engineers for non-technical issues.

5. *WorkPool:* The *WorkPool* role maintains a pool of telephone repair requests. Customers interact with this role to place requests and engineers interact with this role to select tasks to undertake.

6. *Customer:* The *Customer* role models the behaviour of a customer. In involves placing telephone repair requests, receiving relevant information and arranging appointments with field engineers.

7. *Brulebase:* This role maintains a database of business rules. It interacts with manager providing information about the current work policy of the business.

8. *TravelManager:* The *TravelManager* role provides travel information to the field engineer including current location, traffic information and optimal route to next telephone repair task.

9. *TravelinfoBase:* This role store travel information from various travel resources i.e. GPS and traffic databases.

10. *Knowledgefinder:* This role searches for experts and obtains assistance regarding complex work tasks.

11. *Knowledgebase:* The *Knowledgebase* role maintains and manages a database of expertise about telephone repair tasks.

**Specifying design constraints**

In fig. 4 compositional constraints for the roles described are specified in *RCL (Role Constraint Language).*

*RCL* is simple declarative constraint language we introduced to represent design constraints on agent and role characteristics. Any non-obvious use of *RCL* in fig. 5 is described below together with the relevant design constraints. *RCL* itself is described in detail in [8].

```
/* ROLE DEFINITIONS */

role employee, coordinator, mentor,
     customer, travelmanager,
     knowledgefinder;

role workpool, brulebase, workerassistant;
     travelinfobase, knowledgebase {
     int memory;

     workpool.memory = 1;
     brulebase.memory = 1;
     travelinfobase.memory = 2;
     knowledgebase.memory = 2;
     workerassistant.memory = 1;

role manager {
     collaborators = {Coordinator,
                      Brulebase};
     protocols = {contracting};
```

```
/* ROLE CONSTRAINTS */

in(employee, coordinator);
in(employee, manager);

not(customer, employee);
not(customer, travelinfobase);
not(customer, knowledgebase);
not(mentor, manager);
not(manager, coordinator);

and(mentor, employee);

merge(coordinator, travelmanager,
knowledgefinder,
workerassistant);

/* GENERAL CONSTRAINTS */

Constraint Y {
      forall a:Agent {
      a.memory <= 2
      }
}
```

**Figure 4: Compositional constraints for telephone repair service teams roles**

Roles in *RCL* are specified in a manner similar to programming languages. Roles that directly manipulate databases require access to some storage space. This is modelled by the performance variable *memory*. The memory requirements of each role are different. For example, *Travelinfobase* and *Knowledgebase* require twice as much memory as *Workpool* and *Brulebase*.

Part of the definition of the characteristics of the *Manager* role is shown in more detail in fig. 3. The collaborators of the *Manager* role are the *Coordinator* and *Brulebase* roles and its interaction protocol is the Contract Net. The Employee role is contained in both *Manager* and *Coordinator* roles. Furthermore, a *Manager* cannot coexist with *Mentor* or *Coordinator* and for security purposes a *Customer* cannot coexist with *Employee*, *Travelinfobase* or *Knowledgebase*. In order for an agent to be *Mentor* it must also be an *Employee*.

When an agent plays all three *Coordinator*, *TravelManager* and *KnowledgeFinder* roles then overheads occur in synchronising results from these three different activities. This is modelled as a merging of the *Coordinator*, *Travelmanager* and *Knowledgefinder* resulting to the *WorkerAssistant* role. The *WorkerAssistant* role requires some storage space to store intermediate synchronisation results.

An example of non-functional requirements is the limit to the memory each agent could occupy. In this case study, agents supporting field engineers should be able to operate in PDAs with limited amount of memory. This is modelled as a general design constraint on the performance variable *memory.* A possible agent organisation satisfying the above design constraints is depicted in fig. 5.

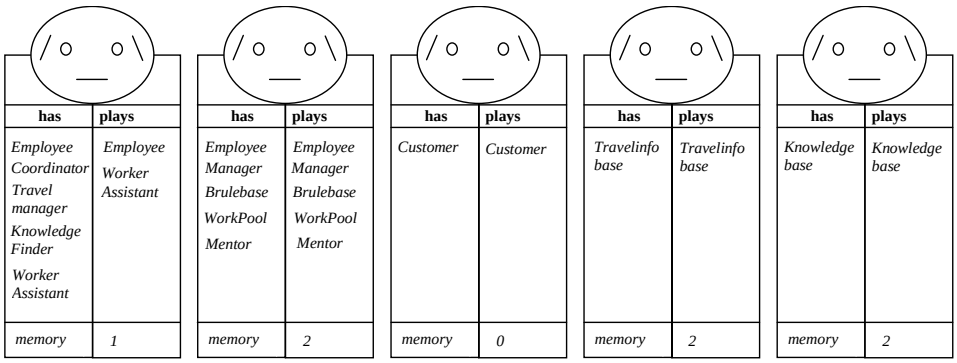


**Figure 5: Agent types for the telephone repair service teams case study**

## Implementation

We have developed an experimental implementation of the system shown in figure 1 that provides support for engineers to describe design concerns in the role algebra described above and then to reuse previous role models in their implementations.
In order to implement these tools we needed to change the development process assumed by the Zeus toolkit.

In Figure 6 we show how role selection and the notion of agent type definition and selection is incorporated into the process at the design phase before realisation and configuration and then deployment.

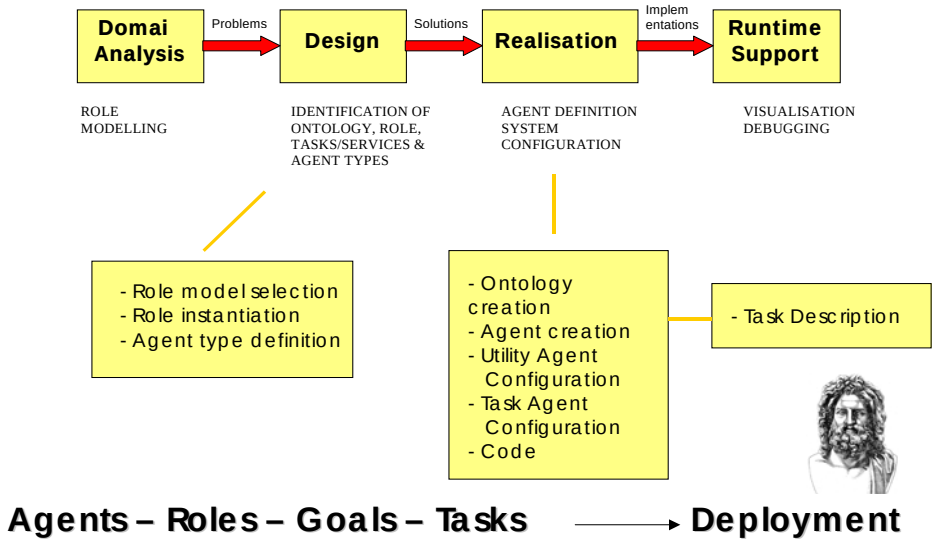


**Figure 6. New Agent Design Process in Zeus**

## Conclusions - further work

Existing approaches to agent organisation design do not pay enough attention to semi-automating the transformation of analysis into design. In this paper, a simple role algebra enabling automatic allocation of roles to agents has been introduced. This approach enables reuse of organisational design settings by representing them as role models being able to be manipulated considering the proposed role algebra.

However, there are issues that have not been addressed yet. For example, agents can play different roles in different contexts and hence the possible contexts should be considered when designing agent organisations. Furthermore, it is planned to use the role algebra to

enable allocating and de-allocating roles to agents dynamically on run-time. This will require alterations to the mechanisms for naming and namespace management that are currently used in Agent standards and agent systems, and may provide support for a open agent type management system.

More importantly than either of these considerations for further action is the need to demonstrate the value of this approach. We have shown that role modelling is a feasible mechanism for recording design knowledge and reusing it at a later date. We have no evidence that our method is a better way of doing this than any other approach, or indeed that it is better than not doing it at all! To provide answers to the questions posed by development teams before they commit to a technology ("how much will it cost", "what sort of training will we need", "what it the return on investment we can expect", "what impact will this have on our development timescales") will require a substantial investigation to measure the impact of industrialised versions of this technology on a number of real world projects. This investigation will have to study the impact on code reuse, productivity, cost and quality of product, and show that the technology can be used by developers with typical skills and abilities.

Before such a study can be mounted a great deal needs to be done, we need to move our implementations beyond the small examples that we have tried so far. We will need to implement tools that are attractive and intuitive for developers and we will need to integrate the technology with widespread defacto standards. Last, but by no means least, we will be required to document and implement a large number of role models to provide our trailists with sufficient material to test the validity of the approach.

## Acknowledgements

This work has been supported by BT under a grant from the office of the Chief Technologist (No. ML816801/MH354166).

## References

[1] Andersen, E., *Conceptual Modelling of Objects: a role modelling approach*, in *Dept of Computer Science*. 1997, University of Oslo: Oslo.

[2] Artikis, A. and J. Pitt, *A Formal Model of Open Agent Societies*, in *Proceedings of Autonomous Agents 2001*. 2001: Montreal. p. 192-193.

[3] Biddle, B.J., *Role Theory: Expectations, Identities and Behaviours*. 1979, London: Academic Press.

[4] Depke, R., R. Heckel, and J.M. Kuster, *Improving the Agent-oriented Modeling Process by Roles*, in *Proceedings of the fifth international conference on Autonomous Agents*. 2001, ACM Press: Montreal, Canada.

[5] Evans, R., *MESSAGE: Methodology for Engineering Systems of Software Agents*. 2000, BT Labs: Ipswich.

[6] Ferber, J. and O. Gutknecht. *A meta-model for the analysis and design of organisations of Multi-Agent systems*. in *International Confernce in Multi-Agent Systems (ICMAS 98)*. 1998. Paris, France: IEEE Press.

[7] Hilaire, V., et al., *Formal Specification and Prototyping of Multi-Agent Systems*, in *Engineering Societies in the Agents' World ESAW'00 (in ECAI'00)*. 2000: Berlin.

[8] Karageorgos, A. and N. Mehandjiev, *Specifying Role Constraints in RCL*. 2001, UMIST: Manchester. p. 35.

[9] Kendall, E.A., *Role models - patterns of agent system analysis and design.* BT Tech. Journal, 1999. **17**(4): p. 46-57.

[10] Kendall, E.A. and L. Zhao. *Capturing and Structuring Goals*. in *Workshop on Use Case Patterns, Object Oriented Programming Systems Languages and Architectures (OOPSLA),*. 1998.

[11] Nwana, H.S., et al., *Zeus: A toolkit for Building Distributed Multi-Agent Systems.* Applied Artificial Intelligence Journal, 1999. **13**(1): p. 187-203.

[12] Omicini, A. *SODA : Societies and Infrastructures in the Analysis and Design o Agent-based Systems*. in *Workshop on Agent-Oriented Software Engineering*. 2000. Limetick, Ireland.

[13] Parunak, V., J. Sauter, and S. Clark, *Toward the Specification and Design of Industrial Synthetic Ecosystems*, in *Intelligent Agents IV: Agent Theories, Architectures, and Languages*, M.P. Singh, A. Rao, and M.J. Wooldridge, Editors. 1998, Springer Verlag: Berlin. p. 45-59.

[14] Scott, W.R., *Organisations: Rational, Natural and Open Systems*. 1992, New York: Prentice Hall International.

[15] So, Y.-p. and E.H. Durfee, *Designing Organisations for Computational Agents*, in *Simulating Organisations: Computational Models of institutions and groups*, M.J. Prietula, K.M. Carley, and L. Gasser, Editors. 1998, AAAI Press. p. 47-64.

[16] Sparkman, C.H., S.A. DeLoach, and A.L. Self. *Automated Derivation of Complex Agent Architectures from Analysis Specifications*. in *Agent-Oriented Software Engineering (AOSE-2001)*. 2001. Montreal, Canada.

[17] Stark, J., et al., *ACSOSS: a case study applying the MESSAGE analysis method*. 2001, BT Labs: Ipswich.

[18] Thompson, S.G. and B.R. Odgers. *Collaborative Personal Agents for Team Working*. in *Proceedings of the AISB Symposium*. 2000. Birmingham, England.

[19] Wooldridge, M., N.R. Jennings, and D. Kinny, *The Gaia methodology for agent-oriented analysis and design*. International Journal of Autonomous Agents and Multi-Agent Systems, 2000. **3**(3): p. 285-312.

[20] Yu, L. and B.F. Schmid. *A Conceptual Framework for Agent Oriented and Role Based Workflow Modelling*. in *CaiSE Workshop Conference on Agent Oriented Information Systems (AOIS´99)*. 1999. Heidelberg: MIT Press.

Zambonelli, F.; Jennings, N.R.; and Wooldridge, M. J. 2000. Organizational Abstractions for the Analysis and Design of Multi-Agent Systems. In *Proceedings of the 1st Workshop on* Agent-Oriented Software Engineering, Springer-Verlag.