Specifying Reuse Concerns in Agent System Design Using a Role Algebra

Anthony Karageorgos[1], Simon Thompson[2], and Nikolay Mehandjiev[1]

[1] Dept. of Computation, UMIST, Manchester M60 1QD, UK
{karageorgos, nikolay}@computer.org
[2] Intelligent Systems Research, BTexact Technologies, Ipswich, UK
simon.2.thompson@bt.com

Abstract. 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 complex "intelligent" services from many different service providers. In this paper a mechanism for recording these decisions is described and grouping functionality into Roles which can then be combined using the recorded design knowledge is subsequent development episodes. We illustrate how design decisions can be captured, discuss the semantics of the constructs we introduce and how these abstractions can then be used as the basis of reuse in an extension of the Zeus agent toolkit.

1 Introduction and Motivation

Multi-agent system architectures can be viewed as organised societies of individual computational entities e.g. [6, 15, 19], and the problem of designing a multi-agent system can be viewed as 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 often 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. [4,7,9,12,20].

The Zeus tool [11] was released as open source software in 1999, and used in a number of projects internally at BT both before and after release. It became clear that a number of factors limited the usefulness of the tool for real world application

development. These problems were especially acute when the tool was used by commercial developers, who had no background in agent development or AI, but prodigious skills with respect to e-business and distributed systems development. One of the key issues that was repeatedly raised by real world developers was the lack of a reuse model for agent code that they had developed. After some analysis the reasons for these problems became clear to us.

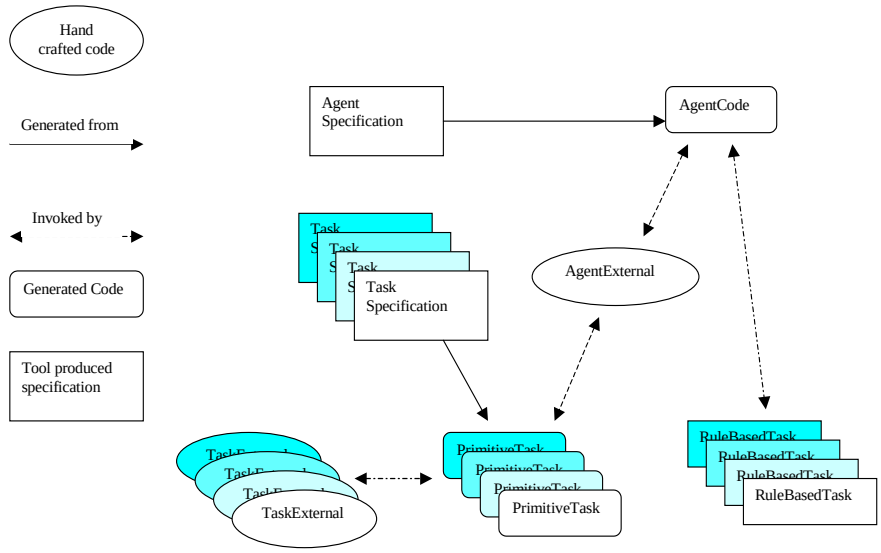

Fig. 1. Modules that define a Zeus Agent

The Zeus system, requires software engineers to produce an "Agent External" (which is called when the agent is run in order to set-up things like database connections and user interfaces), and a set of tasks that the agent uses. The tasks are either "primitive" or "rule-based". Primitive tasks are executed by the planner and rule-based tasks are executed by an expert-system style inference engine. The agent code itself is automatically generated from a specification loaded into a tool. In later versions of Zeus the concept of a "task external" was introduced to allow code to be cleanly added to the task model of the agent, but this exacerbated the proliferation of components that made up the agent model. This problem is illustrated in Figure 1, which shows the relationships between 7 different components that must be defined to create an agent. All of these components are required to support a BDI agent model, and the components use knowledge representations that are appropriate for the use of each (for example production rules for rule based tasks and Java code for user interfaces).

But because BDI agents like Zeus are so complex to define, the design imperative of modularity appears to actually be harmful to developers that are attempting to understand the system and reuse components of it. In Figure 1 we use shading to indicate four modules which are part of the agent developed using modular design. Logic for these modules may reside in any of the components of the agent, and may

also be entangled in the code for the agent external. Zeus developers have no way of indicating what module a component or piece of code is a part of, and because of this, reusing the various subsystems from project to project becomes virtually impossible. It rapidly became clear to us that the standard abstractions for development were deficient when it came to dealing with this problem and that we needed to be able to encode the relationships between components in some way so that they could be separated. We also identified a requirement that the non-functional concerns of the developer in the context of this particular system could be separated from the functional concerns common to any use case for the component.

2 Designing agent organisations

The need to develop agent systems of realistic complexity in a reliable and systematic manner has spawned a number of methodologies for agent-oriented analysis and design such as MESSAGE [5], GAIA [19] and SODA [12]. All these methodologies use a number of analysis and design sub-models, each emphasising a particular analysis or design aspect. These can be further improved in the following ways:

- 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 optimise the agent organisation before it has been deployed. To achieve this, we need to identify some means for considering non-functional requirements before actually deploying the multi-agent system. 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. Non-functional requirements in a system can be elicited by a number of standard techniques, for example by using requirement templates [22].
- By reusing organisational settings. This view regarding reuse of organisational settings has been inspired by the concepts introduced in [21]. It is believed 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.

3 Our Approach

Figure 2 shows an overview of our approach. Our objective is to develop a technology that supports software 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.

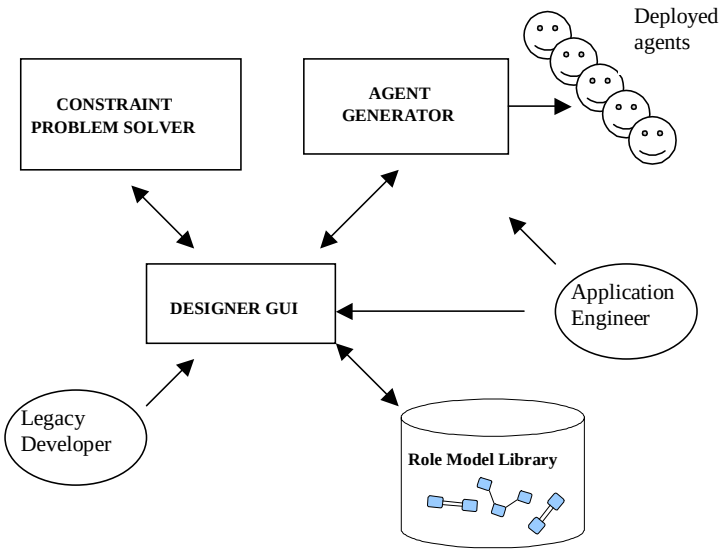

Fig. 2. Conceptual View of Proposed System

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 describes the role modeling abstractions that we have developed in the context of an example derived from a real development project. We then describe how we have implemented a prototype that demonstrates that this knowledge can be used to support design reuse.

3.1. Role-based design

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 them, e.g. [9], investigate the influence of role relations on the design. 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 then 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 various relations 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.

3.2. 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 used 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.

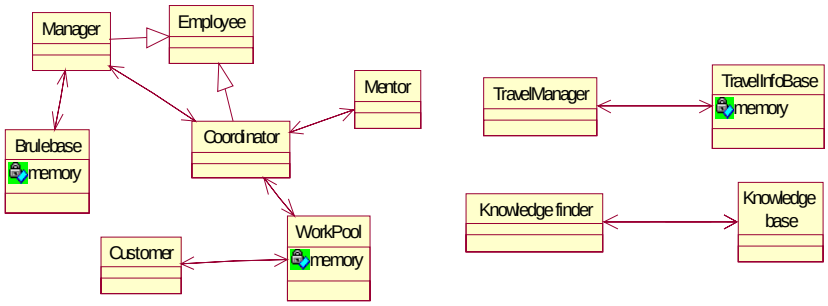

Fig. 3: Role models for the telephone repair service teams case

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.

4 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 knowledge management [17, 18].

Travel management is about supporting mobile workers for moving between repair task locations. 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 about work-related tasks.

4.1. Role identification

To model this system in terms of roles, we need to first 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.*

We can use the methodology from [10], the identify the following roles (Fig. 3):

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:* This role models the team manager. This includes confirming task allocation, monitoring work and ensuring that business rules are followed.
4. *Mentor:* This 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.

4.2. Specifying design constraints

Having identified the roles in our case study, we can proceed to define some constrains regarding how they can be combined in agents. We call these *compositional constraints*, and use a simple *Role Constraint Language (RCL)* to specify them. The compositional constraints relevant to our case study are shown on Fig. 4. *RCL* is simple declarative constraint language we introduced to represent design constraints on agent and role characteristics. Any non-obvious use of *RCL* in Figure 4 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(customer, travelmanager);
not(mentor, manager);
not(manager, coordinator);

and(mentor, employee);

merge(coordinator, travelmanager,
knowledgefinder,
workerassistant);

/* GENERAL CONSTRAINTS */

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

Fig. 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 Figure 4. 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*.

4.3. From Roles To Agents

Having defined our example roles, we need to aggregate them into agents. We define this task as a search problem: we must search the space of all possible system designs in order to find a design that doesn't violate any of the constraints that are defined in the role relationships and satisfies the context that the system is to be used in.

A possible agent organisation satisfying the above design constraints is shown in Fig. 5. The process of allocating roles to agents can be extremely complex because it is necessary to provide designers with options for the number of types of agents that must be produced and to allow them to "hardpin" roles to particular types. However, a simple algorithm illustrates how this process works in general.

1. Allocate all roles to a single agent
2. Make a list of constraints that are violated in the current design
3. Find the role with the most violated constraints
4. If the role can be moved to any agent in the design without violating any constraints, move it to that agent and add to that agent any roles that are *required* by the newly added role.
5. Else create a new agent and move the role to that agent and add any *requires* roles to that agent
6. Repeat 2 to 5 until the list produced at 2 is empty
7. Create the plays list by applying all *mergeswith* relations for each agent

Algorithm 1. Simple role allocation procedure

This algorithm assumes that we prefer agent systems with many roles allocated to few agents. Other algorithms can also be constructed easily, which, for example, prefer designs with many *mergeswith* roles grouped in particular agents. It should be noted that the above procedure and many others that we have considered is not guaranteed to terminate and that the efficiency and scalability of design algorithms of this sort are the topic of ongoing research and are beyond the scope of this paper.

5 A role algebra for agent system design

In order to execute the role allocation algorithm described above, we need to have a formally specified relationships and constraints between roles. We have used role theory [3] and case studies of human activity systems, e.g. [17], to identify six basic role relations, which we have used in the example so far. In this section, we define a formal model of role relations, 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 as described in the previous section.

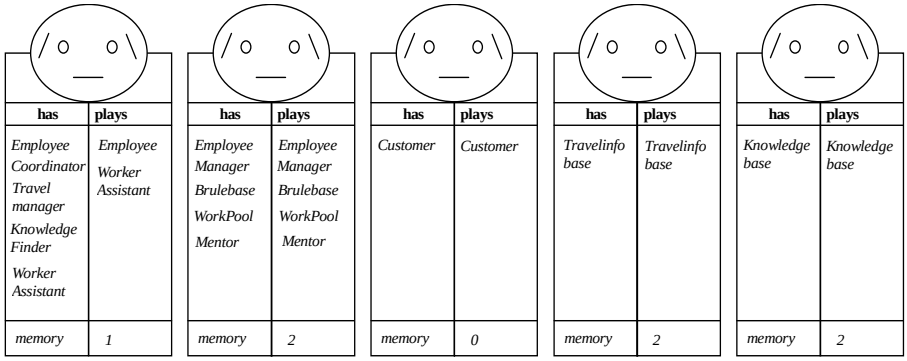

Fig. 5: Agent types for the telephone repair service teams case study

5.1. 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 *Employee* and *Member of staff* can be used to refer to people employed in an organisation. When two roles are equal, an agent playing the first one also plays the other one at the same time. The relation $Equals \subseteq R \times R$ is an equivalence relation since it is reflexive, symmetric and transitive:

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

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. In the mobile working example above we have specified that a customer *excludes* employee not(customer, employee). 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:

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

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

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:

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

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

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. In the mobile working example we specify that a mentor *requires* employee - that in order to act as a mentor you must also be an employee. 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:

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

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

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 *Customer* 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:

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

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

6 **Mergeswith (merge)** — This 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 the mobile working example the roles of Travelmanager, Coordinator and Knowledgefinder merge to Workerassistant because otherwise a synchronisation overhead would be incurred. 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. This relation is used to achieve role synergy. The relation *Mergeswith* $\subseteq R \times R$ is symmetric:

$\forall (r_1, r_2) : R \times R$ (r_1 merge $r_2 \Rightarrow r_2$ merge r_1)

5.2. Semantics of role relations

We describe the semantics of role relations by using a two-sorted algebra (Fig. 6) to represent agent organisation. The algebra includes two sorts, *A* representing agents and *R* representing roles.

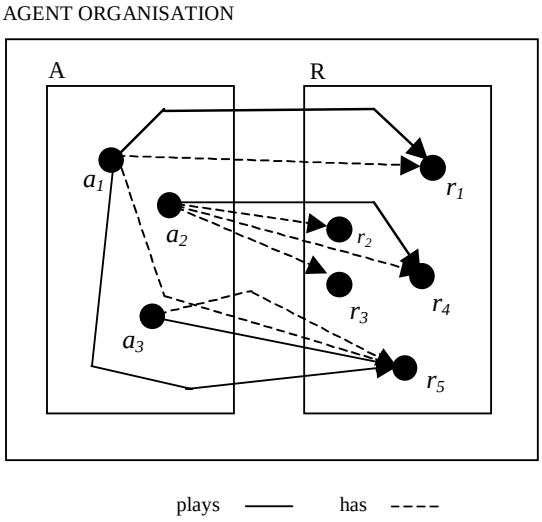

Fig. 6: Semantics of role relations

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 a 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.

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.

6 Implementation

We have developed an experimental implementation of the system shown in Figure 2 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.

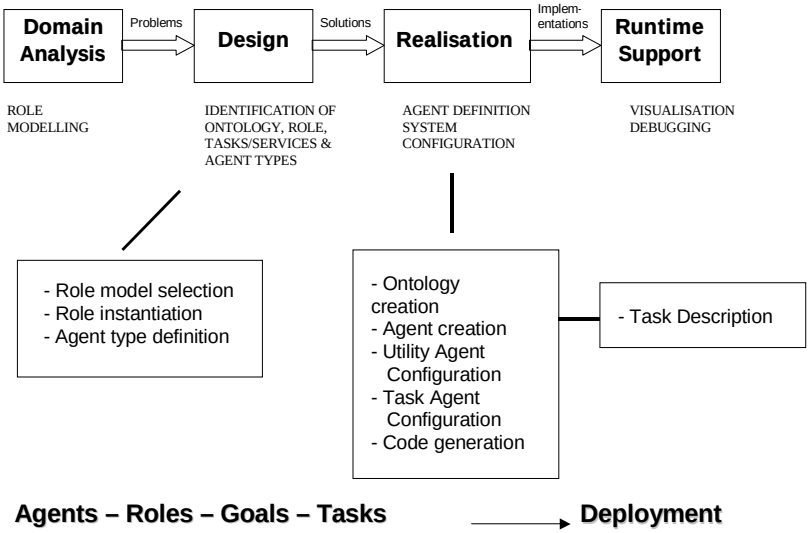

Fig. 7. New Agent Design Process in Zeus

In order to implement the tools shown on Figure 2 we needed to change the development process assumed by the Zeus toolkit. This change is detailed in Figure 7, where 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.

7 Promoting Reuse

In the introduction and motivation section of this paper a discussion of the problems e-commerce developers encountered when they attempted to use Zeus was given, and we emphasized the barrier to adoption that the difficulties in reusing entangled agent application code presents. A description of a system of modularity based on role modeling was then given, along with the mobile working example which shows how this may work in a practical situation. However, we have not addressed how developing agents using role models can promote reuse.

Roles in our model are a set of constraints that are attached to a group of primitive and rule-based tasks and some "external" code which can then be deployed into an agent. In this way the agent concerned is equipped with problem solving knowledge in

the form of new plan atoms (primitive tasks) and reactive behaviors (rule based tasks). In the mobile working example we developed a *travelmanager* role, which contains a set of tasks for planning a route using traffic information and current location. This role could be very useful when we later want to develop a Personal Travel Assistant (PTA).

In a PTA we would not have a *employee* role, but we would have a *customer* role, a *airtravelProvider*, *hotelProvider* and *broker*. The *customer* and *travelmanager* roles are imported from the mobile working role model, and the other roles are developed from scratch. We then specify a set of simple Role Constraints which govern the way these roles combine:

```
not (customer, travelmanager)
not (broker, hotelProvider)
not (broker, travelinfobase)
not (broker, airtravelProvider)
not (broker, customer)
not (hotelProvider, customer)
not (airtravelProvider, customer)
```

When we apply Algorithm 1 to the problem it can be seen that this will lead to a role allocation of

```
A1 :customer
A2: broker, travelmanager
A3: airtravelProvider, hotelProvider
```

We now have a robust and systematic manner to re-use behavioral models (roles and role models) in new applications, and to allocate roles to agents during system design.

8 Conclusions and further work

To enable effective design of complex multi-agent systems, we need to support the complexity involved in this process by semi-automating the transformation of analysis into design and reusing previous design knowledge. This is a weakness of the majority of existing approaches to agent organisation design and a focus of the work reported here. 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.

In this paper we illustrate our approach with an example from the field of collaborative working, however, we have also considered systems from other fields such as e-commerce. Of course, it is an open question as to how widely the roles and methods that we outline can be applied, however we believe that this is a general programming and compilation method which can be used by developers of many different types of distributed system.

Some issues have not been addressed yet and require further work. 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 an 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! Providing 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 of further work is necessary. We need to move our implementations beyond the small examples 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 *de-facto* standards. Last, but by no means least, we will be required to document and implement a large number of role models to provide our trail subjects with sufficient material to test the validity of the approach.

9 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.
[21] 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.
[22] Robertson,J. & Robertson, S. "Volere Requirements Specification Template Edition 8" http://www.guild.demon.co.uk/SpecTemplate8.pdf