A Methodology for Agent-Oriented Analysis and Design

Michael Wooldridge*, Nicholas R. Jennings*, and David Kinny†

*Department of Electronic Engineering
Queen Mary & Westfield College
London E1 4NS, UK
{M.J.Wooldridge, N.R.Jennings}@qmw.ac.uk

†Department of Computer Science
University of Melbourne
Parkville 3052, Australia
dnk@cs.mu.oz.au

Abstract

This paper presents a methodology for agent-oriented analysis and design. The methodology is general, in that it is applicable to a wide range of multi-agent systems, and comprehensive, in that it deals with both the macro-level (societal) and the micro-level (agent) aspects of systems. The methodology is founded on the view of a system as a computational organisation consisting of various interacting roles. We illustrate the methodology through a case study (an agent-based business process management system).

1 Introduction

Progress in software engineering over the past two decades has primarily been made through the development of increasingly powerful and natural abstractions with which to model and develop complex systems. Procedural abstraction, abstract data types, and, most recently, objects, are all examples of such abstractions. It is our belief that agents represent a similar advance in abstraction: they may be used by software developers to more naturally understand, model, and develop an important class of complex distributed systems.

If agents are to realise their potential as a software engineering paradigm, then it is necessary to develop software engineering techniques that are specifically tailored to them. Existing software development techniques (for example, object-oriented analysis and design [1, 5]) will simply be unsuitable for this task. There is a fundamental mismatch between the concepts used by object-oriented developers (and indeed, by other mainstream software engineering paradigms) and the agent-oriented view [20, 22]. In particular, extant approaches fail to adequately capture an agent's flexible, autonomous problem-solving behaviour, the richness of an agent's interactions, and the complexity of an agent system's organisational structures. For these reasons, this paper outlines a methodology that has been specifically tailored to the analysis and design of agent-based systems.

The remainder of this paper is structured as follows. We begin, in the following sub-section, by discussing the characteristics of applications for which we believe our analysis and design methodology is appropriate. Section 2 gives an overview of the main concepts used by the methodology. Agent-based analysis is discussed in section 3, and design in section 4. The methodology is demonstrated by means of a case study in section 5, where we show how it was applied to the design of a real-world agent-based system for business process management [13]. Related work is discussed in section 6, and some conclusions are presented in section 7.

Domain Characteristics

Before proceeding, it is worth commenting on the scope of our work, and in particular, on the characteristics of domains for which we believe the methodology is appropriate. It is intended that the methodology be appropriate for the development of systems such as ADEPT [13] and ARCHON [12]. These are large-scale real-world applications, with the following main characteristics:

- Agents are coarse-grained computational systems, each making use of significant computational resources (think of each agent as having the resources of a UNIX process.)
- It is assumed that the goal is to obtain a system that maximises some global quality measure, but which may be sub-optimal from the point of view of the system components. Our methodology is *not* intended for systems that admit the possibility of true conflict.
- Agents are heterogeneous, in that different agents may be implemented using different programming languages and techniques. We make no assumptions about the delivery platform.
- The overall system contains a comparatively small number of agents (less than 100).

The methodology we propose is comprehensive, in that it deals with both the macro (societal) level and the micro (agent) level aspects of design. It represents an advance over previous agent-oriented methodologies in that it is neutral with respect to both the target domain and the agent architecture (see section 6 for a more detailed comparison).

2 A Conceptual Framework

Our methodology is intended to allow an analyst to go systematically from a statement of requirements to a design that is sufficiently detailed that it can be implemented directly. In applying the methodology, the analyst moves from abstract to increasingly concrete concepts. Each successive move introduces greater implementation bias, and shrinks the space of possible systems that could be implemented to satisfy the original requirements statement.

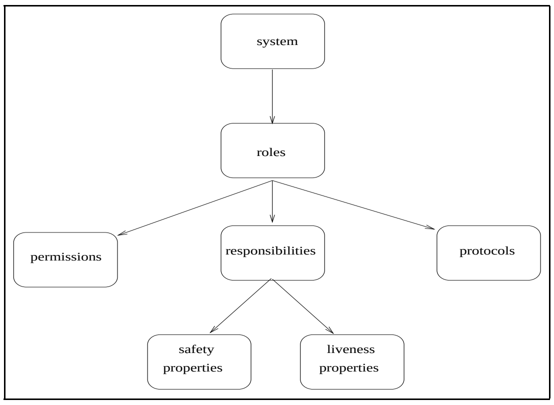

Figure 1: Analysis Concepts

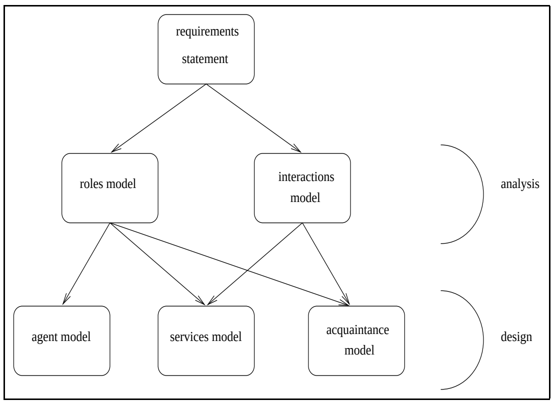

Figure 2: Relationships between the methodology's models

The methodology borrows some terminology and notation from object-oriented analysis and design, (specifically, FUSION [5]). However, it is not simply a naive attempt to apply such methods to agent-oriented development. Rather, our methodology provides an agent-specific set of concepts through which a software engineer can understand and model a complex system. In particular, the methodology encourages a developer to think of building agent-based systems as a process of *organisational design*.

The main concepts can be divided into two categories: *abstract* and *concrete*. Abstract entities are those used during analysis to conceptualise the system, but which do not necessarily have any *direct* realisation within the system. Concrete entities, in contrast, are used within the design process, and will typically have direct counterparts in the run-time system.

The most abstract entity in our concept hierarchy is the *system* — see Figure 1. Although the term "system" is used in its standard sense, it also has a related meaning when talking about an agent-based system, to mean "society" or "organisation". That is, we think of an agent-based system as an artificial society or organisation.

The idea of a system as a society is useful when thinking about the next level in the concept hierarchy: *roles*. It may seem strange to think of a computer system as being defined by a set of roles, but the idea is quite natural when adopting an organisational view of the world. Consider a human organisation such as a typical company. The company has roles such as "president", "vice president", and so on. Note that in a concrete *realisation* of a company, these roles will be *instantiated* with actual individuals: there will be an individual who takes on the role of president, an individual who takes on the role of vice president, and so on. However, the instantiation is not necessarily static. Throughout the company's lifetime, many individuals may take on the role of company president, for example. Also, there is not necessarily a one-to-one mapping between roles and individuals. It is not unusual (particularly in small or informally defined organisations) for one individual to take on many roles. For example, a single individual might take on the role of "tea maker", "mail fetcher", and so on. Conversely, there may be many individuals that take on a single role, e.g., "salesman".

A role is defined by three attributes: *responsibilities*, *permissions*, and *protocols*. *Responsibilities* determine functionality and, as such, are perhaps the key attribute associated with a role. An example responsibility associated with the role of company president might be calling the shareholders meeting every year. Responsibilities are divided into two types: *liveness properties* and *safety properties* [18]. Liveness properties intuitively state that "something good happens". They describe those states of affairs that an agent must bring about, given certain environmental conditions. In contrast, safety properties are invariants. Intuitively, a safety property states that "nothing bad happens" (i.e., that an acceptable state of affairs is maintained across all states of execution). An example might be "ensure the reactor temperature always remains in the range 0-100".

In order to realise responsibilities, a role is usually associated with a set of *permissions*. Permissions are the "rights" associated with a role. The permissions of a role thus identify the resources that are available to that role in order to realise its responsibilities. In the kinds of system that we have typically modelled, permissions tend to be *information resources*. For example, a role might have associated with it the ability to read a particular item of information, or to modify another piece of information. A role can also have the ability to *generate* information.

Finally, a role is also identified with a number of *protocols*, which define the way that it can interact with other roles. For example, a "seller" role might have the protocols "Dutch auction" and "English auction" associated with it.

In summary, analysis and design can be thought of as a process of developing increasingly detailed *models* of the system to be constructed. The main models used in our approach are summarised in Figure 2, and elaborated in sections 3 and 4.

3 Analysis

The objective of the analysis stage is to develop an understanding of the system and its structure (without reference to any implementation detail). In our case, this understanding is captured in the system's *organisation*. In more detail, we view an organisation as a collection of roles, that stand in certain relationships to one another, and that take part in systematic, institutionalised patterns of interactions with other roles. To define an organisation, it therefore suffices to define the roles in the organisation, how these roles relate to one another, and how a role can interact with other roles. The aim of the analysis stage is, therefore, to model the system as a multi-agent organisation in precisely this way. Thus, the organisation model is comprised of two further models: the *roles model* (section 3.1) and the *interaction model* (section 3.2).

3.1 The Roles Model

The roles model identifies the key roles in the system. Here a role can be viewed as an abstract description of an entity's expected function. In other terms, a role is more or less identical to the notion of an *office* in the sense that "prime minister", "attorney general of the United States”, or “secretary of state for Education” are all offices. Such roles (or offices) are characterised by two types of attribute:

- *The permissions/rights associated with the role.*

  A role will have associated with it certain permissions, relating to the type and the amount of resources that can be exploited when carrying out the role. In our case, these aspects are captured in an attribute known as the role’s *permissions*.

- *The responsibilities of the role.*

  A role is created in order to *do* something. That is, a role has a certain functionality. This functionality is represented by an attribute known as the role’s *responsibilities*.

Permissions

The permissions associated with a role have two aspects:

- they identify the resources that can legitimately be used to carry out the role — intuitively, they say what *can* be spent while carrying out the role;
- they state the resource limits within which the role executor must operate — intuitively, they say what *can’t* be spent while carrying out the role.

In general, permissions can relate to any kind of resource. In a human organisation, for example, a role might be given a monetary budget, a certain amount of person effort, and so on. However, in our methodology, we think of resources as relating only to the *information* or *knowledge* the agent has. That is, in order to carry out a role, an agent will typically be able to access certain information. Some roles might generate information; others may need to access a piece of information but not modify it, while yet others may need to modify the information. We recognise that a richer model of resources is required for the future, although for the moment, we restrict our attention simply to information.

We use a formal notation for expressing permissions that is based on the FUSION notation for operation schemata [5, pp26–31]. We illustrate this notation below.

Responsibilities

The *functionality* of a role is defined by its *responsibilities*. These responsibilities can be divided into two categories: *liveness* and *safety* responsibilities.

Liveness responsibilities are those that, intuitively, state that “something good happens”. Liveness responsibilities are so called because they tend to say that “something will be done”, and hence that the agent carrying out the role is still alive. Liveness responsibilities tend to follow certain patterns. For example, the *guaranteed response* type of achievement goal has the form “a request is always followed by a response”. The *infinite repetition* achievement goal has the form “*x* will happen infinitely often”. Note that these types of requirements have been widely studied in the software engineering literature, where they have proven to be necessary for capturing properties of *reactive* systems [18].

In order to illustrate the various concepts associated with roles, we will use a simple running example of a “coffee filler” role — the purpose of this role is to ensure that a coffee pot is kept full of coffee for a group of workers. Examples of liveness responsibilities for a CoffeeFiller role might be:

- whenever the coffee is empty, fill it up;
- whenever fresh coffee is brewed, make sure the workers know about it.

$x.y$	x followed by y	$x \mid y$	x or y occurs
$x*$	x occurs 0 or more times	$x+$	x occurs 1 or more times
x^{ω}	x occurs infinitely often	$[x]$	x is optional
$x \parallel y$	x and y interleaved		

Table 1: Operators for liveness expressions

In our model, an agent’s liveness properties are specified via a *liveness expression*, which defines the “life-cycle” of the role. Liveness expressions are similar to the *life-cycle* expression of FUSION [5], which are in turn essentially regular expressions. Our liveness properties have an additional operator, “ω”, for *infinite repetition* (see Table 1 for more details). They thus resemble ω-regular expressions, which are known to be suitable for representing the properties of infinite computations [20].

Liveness expressions define the potential execution trajectories through the various activities and interactions (i.e., over the protocols) associated with the role. The general form of a liveness expression is:

$$\mathsf{RoleName} = \mathit{expression}$$

where RoleName is the name of the role whose liveness properties are being defined, and *expression* is the liveness expression defining the liveness properties of RoleName. The atomic components of a liveness expression are *protocols* — we define protocols below.

To illustrate liveness expressions, consider again the above-mentioned responsibilities of the CoffeeFiller role:

$$\mathsf{CoffeeFiller} = (\mathsf{Fill}.\mathsf{InformWorkers}.\mathsf{CheckStock}.\mathsf{AwaitEmpty})^{\omega}$$

This expression says that CoffeeFiller consists of executing the protocol Fill, followed by the protocol InformWorkers, followed by the protocols CheckStock and AwaitEmpty. These four protocols are then repeated infinitely often. For the moment, we shall treat the protocols simply as labels for interactions and shall not worry about how they are actually defined (this matter will be returned to in section 3.2).

Complex liveness expressions can be made easier to read by structuring them. A simple example illustrates how this is done:

$$\begin{aligned} \mathsf{CoffeeFiller} &= (\mathsf{All})^{\omega} \\ \mathsf{All} &= \mathsf{Fill}.\mathsf{InformWorkers}.\mathsf{CheckStock}.\mathsf{AwaitEmpty} \end{aligned}$$

In many cases, it is insufficient simply to specify the liveness responsibilities of a role. This is because an agent, carrying out a role, will be required to maintain certain *invariants* while executing. For example, we might require that a particular agent taking part in an electronic commerce application never spends more money than it has been allocated. These invariants are called *safety* conditions, because they usually relate to the absence of some undesirable condition arising.

Safety requirements in our methodology are specified by means of a list of predicates. These predicates are typically expressed over the variables listed in a role’s permissions attribute. Returning to our CoffeeFiller role, an agent carrying out this role will generally be required to ensure that the coffee stock is never empty. We can do this by means of the following safety expression:

- $\mathit{coffeeStock} \geq 0$

By convention, we simply list safety expressions as a bulleted list, each item in the list expressing an individual safety responsibility. It is implicitly assumed that these responsibilities apply across *all*

ROLE SCHEMA:	*name of role*
Description	*short English description of the role*
Protocols	*protocols in which the role plays a part*
Permissions	*“rights” associated with the role*
Responsibilities	
Liveness	*liveness responsibilities*
Safety	*safety responsibilities*

Figure 3: Template for Role Schemata

states of the system execution. If the role is of infinitely long duration (as in the CoffeeFiller example), then the invariants must *always* be true.

It is now possible to precisely define the roles model. A roles model is comprised of a set of *role schemata*, one for each role in the system. A role schema draws together the various attributes discussed above into a single place (Figure 3). An exemplar instantiation is given for the CoffeeFiller role in Figure 4. This schema indicates that CoffeeFiller has permission to read the coffeeMaker parameter (that indicates which coffee machine the role is intended to keep filled), and the coffeeStatus (that indicates whether the machine is full or empty). In addition, the role has permission to change the value coffeeStock.

3.2 The Interaction Model

There are inevitably dependencies and relationships between the various roles in a multi-agent organisation. Indeed, such interplay is central to the way in which the system functions. Given this fact, interactions obviously need to be captured and represented in the analysis phase. In our case, such links between roles are represented in the *interaction model*. This model consists of a set of *protocol definitions*, one for each type of inter-role interaction. Here a protocol can be viewed as an institutionalised pattern of interaction. That is, a pattern of interaction that has been formally defined and abstracted away from any particular sequence of execution steps. Viewing interactions in this way means that attention is focused on the essential nature and purpose of the interaction, rather than on the precise ordering of particular message exchanges (cf. the interaction diagrams of OBJECTORY [5, pp198–203] or the scenarios of FUSION [5]).

Our approach means that a single protocol definition will typically give rise to a number of message interchanges in the run time system. For example, consider an English auction protocol. This involves multiple roles (sellers and bidders) and many potential patterns of interchange (specific price announcements and corresponding bids). However at the analysis stage, such precise instantiation details are unnecessary, and too premature.

In more detail, protocol definitions consist of the following set of attributes:

- *purpose*: brief description of the nature of the interaction (e.g. “information request”, “schedule activity” and “assign task”);
- *initiator*: the role(s) responsible for starting the interaction;
- *responder*: the role(s) with which the initiator interacts;
- *inputs*: information used by the role initiator while enacting the protocol;
- *outputs*: information supplied by/to the protocol responder during the course of the interaction;

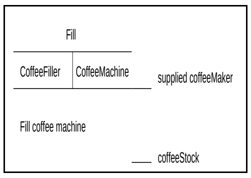

Figure 5: The Fill Protocol Definition

- *processing*: brief description of any processing the protocol initiator performs during the course of the interaction;

By means of an illustration, consider the Fill protocol, which forms part of the CoffeeFiller role (Figure 5). This states that the protocol Fill is initiated by the role CoffeeFiller and involves the role CoffeeMachine. The protocol involves CoffeeFiller putting coffee in the machine named coffeeMaker, and results in CoffeeMachine being informed about the value of coffeeStock. We will see further examples of protocols in section 5.

3.3 The Analysis Process

The analysis stage of the methodology can now be summarised:

1. Identify the *roles* in the system.

   Output: A prototypical roles model — a list of the key roles that occur in the system, each with an informal, unelaborated description.

2. For each role, identify and document the associated *protocols*. Protocols are the patterns of interaction that occur in the system between the various roles.

   Output: An interaction model, which captures the recurring patterns of inter-role interaction.

3. Using the protocol model as a basis, elaborate the roles model.

   Output: A fully elaborated roles model, which documents the key roles occurring in the system, their permissions and responsibilities, and the protocols in which they take part.

4. Iterate stages (1)–(3).

4 Design

The aim of a “classical” design process is to transform the abstract models derived during the analysis stage into models at a sufficiently low level of abstraction that they can be easily implemented. This is not the case with agent-oriented design, however. Rather, our aim is to transform the analysis models into a sufficiently low level of abstraction that traditional design techniques (including object-oriented techniques) may be applied. To put it another way, the agent-oriented analysis and design process is concerned with how a society of agents cooperate to realise the system-level goals, and what is required of each individual agent in order to do this. Actually *how* an agent realises its services is beyond the scope of the methodology, and will depend on the particular application domain.

The design process involves generating three models (see Figure 2). The *agent* model identifies the *agent types* that will make up the system, and the *agent instances* that will be instantiated from these types. The *services model* identifies the main services that will be associated with each agent type. Finally, the *acquaintance model* documents the acquaintances for each agent type.

ROLE SCHEMA:	CoffeeFiller		
DESCRIPTION:	This role involves ensuring that coffee is kept filled, and informing the workers when fresh coffee has been brewed.		
PROTOCOLS:	Fill, InformWorkers, CheckStock, AwaitEmpty		
PERMISSIONS:	reads	supplied coffeeMaker	// *name of coffee maker*
		coffeeStatus	// *full or empty*
	changes	coffeeStock	// *stock level of coffee*
RESPONSIBILITIES			
LIVENESS:	CoffeeFiller = $(\text{Fill.InformWorkers.CheckStock.AwaitEmpty})^{\omega}$		
SAFETY:	• *coffeeStock* ≥ 0		

Figure 4: Schema for role CoffeeFiller

4.1 The Agent Model

The purpose of the agent model is to document the various *agent types* that will be used in the system under development, and the *agent instances* that will realise these agent types at run-time.

An agent type is best thought of as a set of agent roles. There may in fact be a one-to-one correspondence between roles (as identified in the roles model — see section 3.1) and agent types. However, this need not be the case. A designer can choose to package a number of closely related roles in the same agent type for the purposes of convenience. Efficiency will also be a major concern at this stage: a designer will almost certainly want to optimise the design, and one way of doing this is to aggregate a number of agent roles into a single type. An example of where such a decision may be necessary is where the "footprint" of an agent (i.e., its run-time requirements in terms of processor power or memory space) is so large that it is more efficient to deliver a number of roles in a single agent than to deliver a number of agents each performing a single role. There is obviously a trade-off between the *coherence* of an agent type (how easily its functionality can be understood) and the efficiency considerations that come into play when designing agent types. The agent model is defined using a simple *agent type tree*, in which root nodes correspond to roles, (as defined in the roles model), and other nodes correspond to agent types. If an agent type t_1 has children t_2 and t_3, then this means that t_1 is composed of the roles that make up t_2 and t_3.

We document the agent instances that will appear in a system by annotating agent types in the agent model (cf. the qualifiers from FUSION [5]). An annotation n means that there will be exactly n agents of this type in the run-time system. An annotation $m..n$ means that there will be no less than m and no more than n instances of this type in a run-time system ($m < n$). An annotation $*$ means that there will be zero or more instances at run-time, and $+$ means that there will be one or more instances at run-time.

Note that inheritance plays no part in our agent models. Our view is that agents are coarse grained computational systems, and an agent system will typically contain only a small number of roles and types, with often a one-to-one mapping between them. For this reason, we believe that inheritance has no useful part to play in the design of agent types. (However, when it comes to actually implementing agents, inheritance may be used in the normal OO fashion.)

4.2 The Services Model

As its name suggests, the aim of the services model is to identify the *services* associated with each agent role, and to specify the main properties of these services. By a service, we mean a *function* of the agent. In OO terms, a service would correspond to a method; however, we do not mean that services are available for other agents in the same way that an object's methods are available for another object to invoke. Rather, a service is simply a single, coherent block of activity that an agent will engage in.

For each service that may be performed by an agent, it is necessary to document its properties. Specifically, we must identify the *inputs*, *outputs*, *pre-conditions*, and *post-conditions* of each service. Inputs and outputs to services will be derived in an obvious way from the protocols model. Pre- and post-conditions represent constraints on services. These are derived from the safety properties of a role. Note that by definition, each role will be associated with at least one service.

The services that an agent will perform are derived from the list of protocols and responsibilities associated with a role, and in particular, from the liveness definition of a role. For example, returning to the coffee example, there are four protocols associated with this role: Fill, InformWorkers, CheckStock, and AwaitEmpty. There will be at least one service associated with each protocol. In the case of CheckStock, the service (which may have the same name), will take as input the stock level and some threshold value, and will simply compare the two. The pre- and post-conditions will both state that the coffee stock level is greater than 0 – this condition is one of the safety conditions of the CoffeeFiller.

The services model does *not* prescribe an implementation for the services it documents. The developer is free to realise the services in any implementation framework deemed appropriate. For example, it may be decided to implement services directly as methods in an object-oriented language. Alternatively, a service may be decomposed into a number of methods.

4.3 The Acquaintance Model

The final design model is probably the simplest: the *acquaintance model*. Acquaintance models simply define the communication links that exist between agent types. They do *not* define what messages are sent or when messages are sent — they simply indicate that communication pathways exist. In particular, the purpose of an acquaintance model is to identify any potential communication bottlenecks, which may cause problems at run-time (see section 5 for an example). It is generally regarded as good practice to ensure that systems are loosely coupled, and the acquaintance model can help in doing this. On the basis of the acquaintance model, it may be found necessary to revisit the analysis stage and rework the system design to remove such problems.

An agent acquaintance model is simply a graph, with nodes in the graph corresponding to agent types and arcs in the graph corresponding to communication pathways. Agent acquaintance models are *directed* graphs, and so an arc $a \rightarrow b$ indicates that a will send messages to b, but not necessarily that b will send messages to a. An acquaintance model may be derived in a straightforward way from the roles, protocols, and agent models.

4.4 The Design Process

The design stage of the methodology can now be summarised:

1. Create an *agent model*:
   - aggregate roles into *agent types*, and refine to form an agent type hierarchy;
   - document the instances of each agent type using instance annotations.
2. Develop a services model, by examining protocols and safety and liveness properties of roles.
3. Develop an *acquaintance model* from the interaction model and agent model.

5 A Case Study: Business Process Management

This section briefly illustrates how the methodology can be applied, through a case study of the analysis and design of an agent-based system for managing a British Telecom business process (see [13] for more details). For reasons of brevity, we omit some details, and aim instead to give a general flavour of the analysis and design.

The particular application is providing customers with a quote for installing a network to deliver a particular type of telecommunications service. This activity involves the following departments: the *customer service division* (CSD), the *design division* (DD), the *legal division* (LD) and the various organisations who provide the out-sourced service of *vetting customers* (VCs). The process is initiated by a customer contacting the CSD with a set of requirements. In parallel to capturing the requirements, the CSD gets the customer vetted. If the customer fails the vetting procedure, the quote process terminates. Assuming the customer is satisfactory, their requirements are mapped against the service portfolio. If they can be met by a standard off-the-shelf item then an immediate quote can be offered. In the case of bespoke services, however, the process is more complex. DD starts to design a solution to satisfy the customer's requirements and whilst this is occurring LD checks the legality of the proposed service. If the desired service is illegal, the quote process terminates. Assuming the requested service is legal, the design will eventually be completed and costed. DD then informs CSD of the quote. CSD, in turn, informs the customer. The business process then terminates.

Moving from this behavioural description of the system's operation to an organisational view is comparatively straightforward. In many cases there is a one to one mapping between departments and roles. Thus, the VC's, the LD's, and the DD's behaviour are covered by the roles CustomerVetter, LegalAdvisor, and NetworkDesigner respectively. CSD's behaviour falls into two distinct roles: one acting as an interface to the customer (CustomerHandler), and one overseeing the process inside the organisation (QuoteManager). The final role is that of the Customer who requires the quote. Figure 6 defines the role QuoteManager — we omit other role definitions in the interests of brevity. The definition of the VetCustomer protocol is given in Figure 7.

Turning to the design stage, an agent model for this application is given in Figure 8. As this figure illustrates, the implemented system contains five agent types, with two roles (CustomerHandler and QuoteManager) being aggregated into agent type CustomerService-DivisionAgent. The acquaintance model for this domain is defined in Figure 9. (We omit the services model in the interests of brevity.)

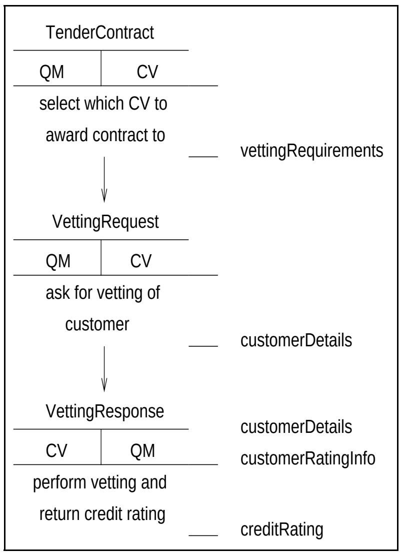

Figure 7: Definition of Protocol VetCustomer between Roles QuoteManager (QM) and CustomerVetter (CV)

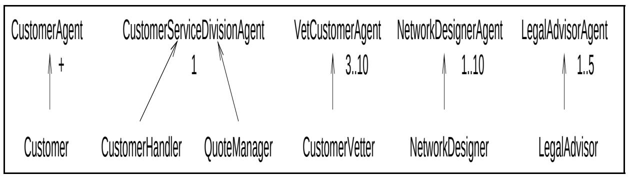

Figure 8: Agent Model for Business Process Management

6 Related Work

As a result of the development and application of robust agent technologies, there has been a surge of interest in agent-oriented methodologies and modelling techniques in the last few years. Many approaches, such as [3, 16] take existing OO modelling techniques or methodologies as their basis, seeking either to extend and adapt the models and define a methodology for their use, or to directly extend the applicability of OO methodologies and techniques, such as design patterns, to the design of agent systems. Other approaches build upon and extend methodologies and modelling techniques from software and knowledge engineering, or provide formal, compositional modelling languages [2] suitable for the verification of system structure and function. A valuable survey can be found in [10].

These approaches usually do not attempt to unify the analysis and design of a MAS with its design and implementation within a particular agent technology. They either regard the output of the analysis and design process as an abstract specification to which traditional lower-level design methodologies may be applied (as proposed in this paper), or else they allow some architectural commitment to be made during analysis or design, but fall short of a full elaboration of the design within the chosen framework. Of the approaches mentioned above, only the AOM approach of [16, 15] makes a strong commitment to a particular agent architecture and proposes a design elaboration and refinement process that leads to directly executable agent specifications. Given the proliferation of available agent technologies, there are clearly advantages to a more

ROLE SCHEMA: QuoteManager (QM)

DESCRIPTION:

Responsible for enacting the quote process. Generates a quote or returns nil if customer inappropriate or service is illegal

PROTOCOLS:

VetCustomer, GetCustomerRequirements, CostStandardService, CheckServiceLegality, CostBespokeService

PERMISSIONS:

reads	supplied customerDetails	// *customer contact information*
	customerRequirements	// *detailed service requirements*
	creditRating	// *customer's credit rating*
	serviceIsLegal	// *boolean for bespoke requests*
generates	quote	// *completed quote or nil*

RESPONSIBILITIES

LIVENESS:

$$\begin{aligned}\text{QuoteManager} &= \text{QuoteResponse}\\ \text{QuoteResponse} &= (\text{VetCustomer} \parallel \text{GetCustomerRequirements}) \mid (\text{VetCustomer} \parallel \text{GetCustomerRequirements}).\text{CostService}\\ \text{CostService} &= \text{CostStandardService} \mid (\text{CheckServiceLegality} \parallel \text{CostBespokeService})\end{aligned}$$

SAFETY:

- $bad(creditRating) \Rightarrow Quote = nil$
- $serviceIsLegal = false \Rightarrow Quote = nil$

Figure 6: Schema for role QuoteManager

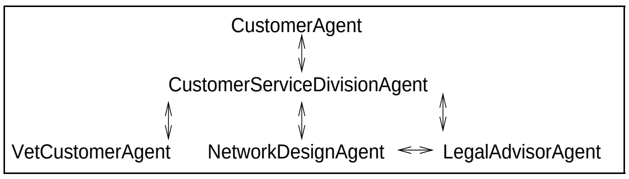

Figure 9: Acquaintance Model for Business Process Management

general approach, as proposed here. However, a disadvantage may be a need for iteration of the entire process if the lower-level design process reveal issues that are best resolved at the agent-oriented level.

Despite this difference in scope, there are many similarities between the AOM approach and that proposed here. The former was developed to fulfill the need for a principled approach to the specification of complex multi-agent systems based on the belief-desire-intention (BDI) technology of the Procedural Reasoning System (PRS) and the Distributed Multi-Agent Reasoning System (DMARS) [17, 6].

The AOM methodology takes as its starting point object-oriented modelling techniques, as exemplified by [19, 1], and adapts and extends them with agent concepts. The methodology itself is aimed at the construction of a set of models which, when fully elaborated, define an agent system specification. The main separation in the models developed is between the *external* and *internal* models. The external models present a system-level view: the main components visible in these models are agents themselves, and they are primarily concerned with agent relationships and interactions, including inheritance and aggregation relationships that allow abstraction of agent structure. In contrast, the internal models which are associated with each distinct agent class are entirely concerned with the internals of agents: their beliefs, goals, and plans.

There are two primary external models; the *agent model*, which describes agent classes and instances, and the *interaction model*, which captures communications and control relationships between agents. The agent model is further divided into an *agent class model* and an *agent instance model*. These two models define the agent classes and instances that can appear, and relate these to one another via inheritance, aggregation, and instantiation relations. Agent classes define various attributes possessed by agents, and amongst these are attributes defining the agent's sets of beliefs, goals, and plans. The analyst is able to define how these attributes are overridden during inheritance. For example, it is assumed that by default, inherited plans have lower priority than those in subclasses. The analyst may tailor these properties as desired.

The internal models, which represent the beliefs, goals and plans of particular agent classes, are direct extensions of OO object models (beliefs, goals) and dynamic models (plans). Thus, for example, an object model is used to describe the objects about which an agent will have beliefs, and the properties of those beliefs, such as whether they have open- or closed-world semantics. Dynamic models, extended with notions of failure and various other attributes, are used to directly represent an agent's plans. These models are thus quite specific to the BDI architecture employed in DMARS. By contrast, the external models are applicable to any BDI agent architecture. The methodology is aimed at elaborating the models described above.

A particular feature of the methodology is its emphasis on the use of *abstract agent classes* as the means to group roles during analysis and model refinement, which permits decisions about the boundaries of concrete agents to be deferred to a late stage of the design process.

Note that the analysis process will be iterative, as in traditional methodologies. The outcome will be a model that comprises specifications in the form required by the DMARS agent architecture. As a result, the move from end-design to implementation using DMARS is relatively simple.

It can be seen that there are many similarities between the AOM external models and the models proposed in this paper. However, the notion of responsibility used in the AOM models is quite informal: safety and liveness requirements are not made explicit at an abstract level, and they lack the notion of permissions used to capture resource usage, which is instead captured implicitly by the belief structure of individual agents. By contrast, the protocols that define the permitted interactions between agents may be developed to a greater degree of detail within the AOM approach, for example as in [14], whereas here protocols are employed as more generic descriptions of behaviour that may involve entities not modelled as agents, such as the coffee machine. Another significant difference is the use in AOM of inheritance between agent classes which is not permitted by the methodology proposed here, as it is of limited value without a specific architectural commitment.

The definition and use of various notions of role, responsibility, interaction, team and society or organization in particular methods for agent-oriented analysis and design has also inherited or adapted much from more general uses of these concepts within multi-agent systems, including organization-focussed approaches such as [9, 7, 11] and sociological approaches such as [4]. However, it is beyond the scope of this paper to compare our definition and use of these concepts with this heritage.

7 Conclusions and Further Work

In this paper, we have described a methodology we have developed for the analysis and design of agent-based systems. The key concepts in this methodology are roles, which have associated with them responsibilities, permissions, and protocols. Roles can interact with one another in certain institutionalised ways, which are defined in the protocols of the respective roles.

There are many issues remaining for future work. Perhaps most importantly, our methodology does not attempt to deal with truly *open* systems, in which agents may not share common goals. This class of systems represents arguably the most important application area for multi-agent systems, and it is therefore essential that our methodology should be able to deal with it. Another aspect of agent-based analysis and design that requires more work is the notion of an organisational structure. At the moment, such structures are only *implicitly* defined within our methodology — within the role and interaction models. However, direct, explicit representations of such structures will be of value for some applications. For example, if agents are used to model large organisations, then these organisations will have an explicitly defined structure. Representing such structures may be the only way of adequately capturing and understanding the organisation's communication and control structures. More generally, the development of *organisation design patterns* might be useful for reusing successful multi-agent system structures (cf. [8]). Finally, we believe that a successful methodology is one that is not only of pragmatic value, but one that also has a well-defined, unambiguous formal semantics. While the typical developer need never even be aware of the existence of such a semantics, it is nevertheless essential to have a precise understanding of what the concepts and terms in a methodology mean [21].

References

[1] G. Booch. *Object-Oriented Analysis and Design (second edition)*. Addison-Wesley: Reading, MA, 1994.

[2] F. Brazier, B. Dunin-Keplicz, N. R. Jennings, and J. Treur. Formal specification of multi-agent systems: a real-world case. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95)*, pages 25–32, San Francisco, CA, June 1995.

[3] Birgit Burmeister. Models and methodologies for agent-oriented analysis and design. In Klaus Fischer, editor, *Working Notes of the KI'96 Workshop on Agent-Oriented Programming and Distributed Systems*. 1996. DFKI Document D-96-06.

[4] C. Castelfranchi. Commitments: from individual intentions to groups and organizations. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95)*, pages 41–48, San Francisco, CA, June 1995.

[5] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes. *Object-Oriented Development: The* FUSION *Method*. Prentice Hall International: Hemel Hempstead, England, 1994.

[6] M. d'Inverno, D. Kinny, M. Luck, and M. Wooldridge. A formal specification of dMARS. In M. P. Singh, A. Rao, and M. J. Wooldridge, editors, *Intelligent Agents IV (LNAI Volume 1365)*, pages 155–176. Springer-Verlag: Berlin, Germany, 1997.

[7] Jacques Ferber and Olivier Gutknecht. A meta-model for the analysis and design of organizations in multi-agent systems. In *Proceedings of the Third International Conference on Multi-Agent Systems (ICMAS-98)*, pages 128–135, Paris, France, 1998.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley: Reading, MA, 1995.

[9] L. Gasser, C. Braganza, and N. Hermann. MACE: A flexible testbed for distributed AI research. In M. Huhns, editor, *Distributed Artificial Intelligence*, pages 119–152. Pitman Publishing: London and Morgan Kaufmann: San Mateo, CA, 1987.

[10] C. A. Iglesias, M. Garijo, and J. C. Gonzalez. A survey of agent-oriented methodologies. In J. P. Müller, M. P. Singh, and A. S. Rao, editors, *Intelligent Agents V — Proceedings of the Fifth International Workshop on Agent Theories, Architectures, and Languages (ATAL-98)*, Lecture Notes in Artificial Intelligence. Springer-Verlag, Heidelberg, 1999.

[11] Toru Ishida, Les Gasser, and Makoto Yokoo. Organization self design of production systems. *IEEE Transactions on Knowledge and Data Engineering*, 4(2):123–134, April 1992.

[12] N. R. Jennings, J. Corera, I. Laresgoiti, E. H. Mamdani, F. Perriolat, P. Skarek, and L. Z. Varga. Using ARCHON to develop real-world DAI applications for electricity transportation management and particle acceleration control. *IEEE Expert*, 11(6):60–88, December 1996.

[13] N. R. Jennings, P. Faratin, M. J. Johnson, T. J. Norman, P. O'Brien, and M. E. Wiegand. Agent-based business process management. *International Journal of Cooperative Information Systems*, 5(2-3):105–130, 1996.

[14] D. Kinny. The AGENTIS agent interaction model. In J. P. Müller, M. P. Singh, and A. S. Rao, editors, *Intelligent Agents V — Proceedings of the Fifth International Workshop on Agent Theories, Architectures, and Languages (ATAL-98)*, Lecture Notes in Artificial Intelligence. Springer-Verlag, Heidelberg, 1999.

[15] D. Kinny and M. Georgeff. Modelling and design of multi-agent systems. In J. P. Müller, M. Wooldridge, and N. R. Jennings, editors, *Intelligent Agents III (LNAI Volume 1193)*, pages 1–20. Springer-Verlag: Berlin, Germany, 1997.

[16] D. Kinny, M. Georgeff, and A. Rao. A methodology and modelling technique for systems of BDI agents. In W. Van de Velde and J. W. Perram, editors, *Agents Breaking Away: Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World, (LNAI Volume 1038)*, pages 56–71. Springer-Verlag: Berlin, Germany, 1996.

[17] David Kinny. *The Distributed Multi-Agent Reasoning System Architecture and Language Specification*. Australian AI Institute, Level 6, 171 La Trobe Street, Melbourne, Australia, 1993.

[18] A. Pnueli. Specification and development of reactive systems. In *Information Processing 86*. Elsevier Science Publishers B.V.: Amsterdam, The Netherlands, 1986.

[19] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliifs, NJ, 1991.

[20] M. Wooldridge. Agent-based software engineering. *IEE Proceedings on Software Engineering*, 144(1):26–37, February 1997.

[21] M. Wooldridge and N. R. Jennings. Intelligent agents: Theory and practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.

[22] M. Wooldridge and N. R. Jennings. Pitfalls of agent-oriented development. In *Proceedings of the Second International Conference on Autonomous Agents (Agents 98)*, pages 385–391, Minneapolis/St Paul, MN, May 1998.