Agent-Based Computing: Promise and Perils

Nicholas R. Jennings
Dept. Electronic Engineering, Queen Mary & Westfield College,
University of London, London E1 4NS, UK.
n.r.jennings@qmw.ac.uk

Abstract

Agent-based computing represents an exciting new synthesis both for Artificial Intelligence (AI) and, more generally, Computer Science. It has the potential to significantly improve the theory and the practice of modelling, designing, and implementing complex systems. Yet, to date, there has been little systematic analysis of what makes an agent such an appealing and powerful conceptual model. Moreover, even less effort has been devoted to exploring the inherent disadvantages that stem from adopting an agent-oriented view. Here both sets of issues are explored. The standpoint of this analysis is the role of agent-based software in solving complex, real-world problems. In particular, it will be argued that the development of robust and scalable software systems requires autonomous agents that can complete their objectives while situated in a dynamic and uncertain environment, that can engage in rich, high-level social interactions, and that can operate within flexible organisational structures.

1 Introduction

An increasing number of computer systems are being viewed in terms of autonomous agents. Agents are being espoused as a new theoretical model of computation that more closely reflects current computing reality than Turing Machines [Wegner, 1997]. Agents are being advocated as the next generation model for engineering complex, distributed systems [Wooldridge, 1997]. Agents are also being used as an overarching framework for bringing together the component AI sub-disciplines that are necessary to design and build intelligent entities [Russell and Norvig, 1995]. Despite this intense interest, a number of fundamental questions about the nature and the use of agents remain unanswered. In particular:

- what is the essence of agent-based computing?
- what makes agents an appealing and powerful conceptual model?
- what are the drawbacks of adopting an agent-oriented approach?
- what are the wider implications for AI of agent-based computing?

These questions can be tackled from many different perspectives, ranging from the philosophical to the pragmatic. This paper proceeds from the standpoint of using agent-based software to solve complex, real-world problems. However in the course of this analysis, a number of broader points are made about the general direction and emphasis of future AI research.

Building high quality software for complex, real-world applications is difficult. Indeed, it has been argued that such developments are one of the most complex construction tasks humans undertake (both in terms of the number and the flexibility of the constituent components and in the complex way in which they are interconnected). Moreover, this statement is true no matter what models and techniques are applied: it is a consequence of the "essential complexity of software" [Brooks, 1995]. Such complexity manifests itself in the fact that software has a large number of parts that have many interactions [Simon, 1996]. Given this state of affairs, the role of software engineering is to provide models and techniques that make it easier to handle this complexity. To this end, a wide range of software engineering paradigms have been devised (e.g. object-orientation [Booch, 1994; Meyer, 1988], component-ware [Szyperski, 1998], design patterns [Gamma *et al.*, 1995] and software architectures [Buschmann *et al.*, 1998]). Each successive development either claims to make the engineering process easier or to extend the complexity of applications that can feasibly be built. Although evidence is emerging to support these claims, researchers continue to strive for more efficient and powerful techniques, especially as solutions for ever more demanding applications are sought.

In this article, it is argued that although current methods are a step in the right direction, when it comes to developing complex, distributed systems they fall short in three main ways: (i) the basic building blocks are too fine grained; (ii) the interactions are too rigidly defined; or (iii)insufficient mechanisms are available for dealing with organisational structure. Furthermore, it will be argued that: *agent-oriented* *approaches can significantly enhance our ability to model, design and build complex (distributed) software systems*.

The remainder of the paper is structured as follows. Section 2 discusses the essence of agent-based computing. Section 3 makes the case for an agent-oriented approach to software engineering. Section 4 provides a brief case study to back up the paper's main arguments. Finally, section 5 outlines an approach for tackling some of the key open problems that need to be addressed if agent-based computing is to reach its full potential.

2 The Essence of Agent-Based Computing

The first step in arguing for an agent-oriented approach to software engineering is to precisely identify and define the key concepts of agent-oriented computing. Here the key definitional problem relates to the term "*agent*". At present, there is much debate, and little consensus, about exactly what constitutes agenthood. However, an increasing number of researchers find the following characterisation useful:

> *an agent is an encapsulated computer system that is situated in some environment, and that is capable of flexible, autonomous action in that environment in order to meet its design objectives* [Wooldridge, 1997]

There are a number of points about this definition that require further explanation. Agents are: (i) clearly identifiable problem solving entities with well-defined boundaries and interfaces; (ii) situated (embedded) in a particular environment—they receive inputs related to the state of their environment through sensors and they act on the environment through effectors; (iii) designed to fulfill a specific purpose—they have particular objectives (goals) to achieve; (iv) autonomous—they have control both over their internal state and over their own behaviour; (v) capable of exhibiting flexible problem solving behaviour in pursuit of their design objectives—they need to be both reactive (able to respond in a timely fashion to changes that occur in their environment) and proactive (able to opportunistically adopt new goals) [Wooldridge and Jennings, 1995].

When adopting an agent-oriented view of the world, it soon becomes apparent that most problems require or involve multiple agents, to represent the decentralised nature of the problem, the multiple loci of control, the multiple perspectives, or the competing interests. Moreover, the agents will need to interact with one another, either to achieve their individual objectives or to manage the dependencies that ensue from being situated in a common environment. These interactions can vary from simple information interchanges, to requests for particular actions to be performed and on to cooperation, coordination and negotiation in order to arrange inter-dependent activities. However, whatever the nature of the social process there are two points that qualitatively differentiate agent interactions from those that occur in other software engineering paradigms. Firstly, agent-oriented interactions occur through a high level (declarative) agent communication language. Consequently, interactions are conducted at the knowledge level [Newell, 1982]: in terms of which goals should be followed, at what time, and by whom (cf. method invocation or function calls that operate at a purely syntactic level). Secondly, as agents are flexible problem solvers, operating in an environment over which they have only partial control and observability, interactions need to be handled in a similarly flexible manner. Thus, agents need the computational apparatus to make context-dependent decisions about the nature and scope of their interactions and to initiate (and respond to) interactions that were not foreseen at design time.

In most cases, agents act to achieve objectives either on behalf of individuals/companies or as part of some wider problem solving initiative. Thus, when agents interact there is typically some underpinning organisational context. This context defines the nature of the relationship between the agents e.g. they may be peers working together in a team or one may be the manager of the other agents. In any case, this context influences an agent's behaviour. Thus it is important to explicitly represent the relationship. In many cases, relationships are subject to ongoing change: social interaction means existing relationships evolve and new relations are created. The temporal extent of relationships can also vary significantly, from just long enough to deliver a particular service once, to a permanent bond. To cope with this variety and dynamic, agent researchers have: devised protocols that enable organisational groupings to be formed and disbanded; specified mechanisms to ensure groupings act together in a coherent fashion; and developed structures to characterise the macro behaviour of collectives [Jennings and Wooldridge, 1998; Wooldridge and Jennings, 1995].

Drawing these points together (figure 1), the essential concepts of agent-based computing are: agents, high level interactions and organisational relationships.

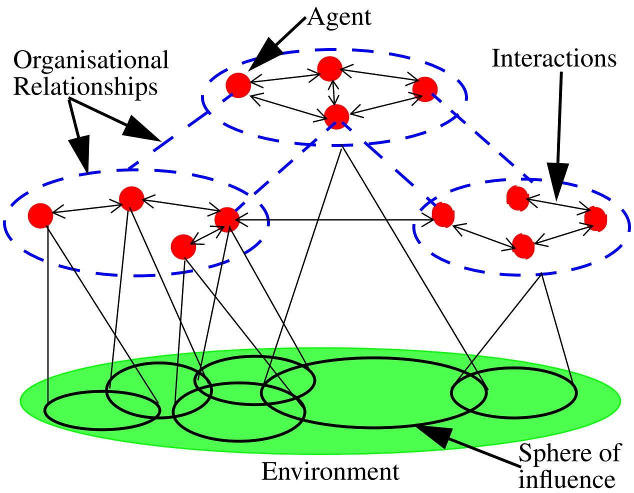

Figure 1: Canonical view of an agent-based system

3 Agent-Oriented Software Engineering

The most compelling argument that can be made for adopting an agent-oriented approach to software development is to have a set of quantitative data that showed, on a standard set of software metrics, the superiority of the agent-based approach over a range of other techniques. However such data does not exist. Hence arguments must be qualitative in nature.

The structure of the argument that will be used here is as follows. On one hand, there are a number of well-known techniques for tackling complexity in software. Also the nature of complex software systems is (reasonably) well understood. On the other hand, the key characteristics of the agent-based paradigm have been elucidated. Thus an argument can be made by examining the degree of match between these two perspectives.

Before this argument can be made, however, the techniques for tackling complexity in software need to be introduced. Booch [1994] identifies three such tools:

- *Decomposition:* The most basic technique for tackling large problems is to divide them into smaller, more manageable chunks each of which can then be dealt with in relative isolation. This helps tackle complexity because it limits the designer's scope; at any given instant only a portion of the problem needs to be considered.
- *Abstraction:* The process of defining a simplified model of the system that emphasises some of the details or properties, while suppressing others. Again, this technique works because it limits the designer's scope of interest at a given time. Attention can be focused on the salient aspects of the problem, at the expense of the less relevant details.
- *Organisation*[1]: The process of identifying and managing the inter-relationships between the various problem solving components. The ability to specify and enact organisational relationships helps designers tackle complexity in two ways. Firstly, by enabling a number of basic components to be grouped together and treated as a higher-level unit of analysis (e.g. the individual components of a sub-system can be treated as a single unit by the parent system). Secondly, by providing a means of describing the high-level relationships between various units (e.g. a number of components may work together to provide a particular functionality).

Next, the characteristics of complex systems need to be enumerated [Simon, 1996]:

- Complexity frequently takes the form of a hierarchy. That is, a system that is composed of inter-related sub-systems, each of which is in turn hierarchic in structure, until the lowest level of elementary sub-system is reached. The precise nature of these organisational relationships varies between sub-systems, however some generic forms (such as client-server, peer, team, etc.) can be identified. These relationships are not static: they often vary over time.
- The choice of which components in the system are primitive is relatively arbitrary and is defined by the observer's aims and objectives.
- Hierarchic systems evolve more quickly than non-hierarchic ones of comparable size. In other words, complex systems will evolve from simple systems more rapidly if there are *stable intermediate forms*, than if there are not.
- It is possible to distinguish between the interactions *among* sub-systems and the interactions *within* sub-systems. The latter are both more frequent (typically at least an order of magnitude more) and more predictable than the former. This gives rise to the view that complex systems are *nearly decomposable*: sub-systems can be treated almost as if they are independent of one another, but not quite, since there are some interactions between them. Moreover, although many of these interactions can be predicted at design time, some cannot.

With these two characterisations in place, the form of the argument can be expressed:

- Show agent-oriented decomposition is an effective way of partitioning the problem space of a complex system (section 3.1);
- Show that the key abstractions of the agent-oriented mindset are a natural means of modelling complex systems (section 3.2);
- Show the agent-oriented philosophy for dealing with organisational relationships is appropriate for complex systems (section 3.3);

Having made the case that agents are well suited for engineering complex systems, a number of pragmatic issues that will affect whether agents catch on as a software engineering paradigm are examined (section 3.4). Finally, the downside of agent-oriented developments is discussed (section 3.5).

3.1 Merits of Agent-Oriented Decomposition

Complex systems consist of a number of related sub-systems organised in a hierarchical fashion. At any given level, sub-systems work together to achieve the functionality of their parent system. Moreover, within a sub-system, the constituent components work together to deliver the overall functionality. Thus, the same basic model of interacting components, working together to achieve particular objectives, occurs throughout the system.

Given this fact, it is entirely natural to modularise the components in terms of the objectives they achieve[2]. In other words, each component can be thought of as achieving one or more objectives. A second important observation is that

[1] Booch [1994] actually uses the term "hierarchy" for this final point. However, the more neutral term "organisation" is used here.

complex systems have multiple loci of control: “real systems have no top” [Meyer, 1988] pg 47. Applying this philosophy to objective-achieving decompositions means the individual components should localise and encapsulate their own control. Thus, entities should have their own thread of control (i.e. they should be active) and they should have control over their own actions (i.e. they should be autonomous).

For the active and autonomous components to fulfil both their individual and collective objectives, they need to interact with one another (recall complex systems are only nearly decomposable). However the system’s inherent complexity means it is impossible to know *a priori* about all potential links: interactions will occur at unpredictable times, for unpredictable reasons, between unpredictable components. For this reason, it is futile to try and predict or analyse all the possibilities at design-time. It is more realistic to endow the components with the ability to make decisions about the nature and scope of their interactions at run-time. From this, it follows that components need the ability to initiate (and respond to) interactions in a flexible manner.

The policy of deferring to run-time decisions about component interactions facilitates the engineering of complex systems in two ways. Firstly, problems associated with the coupling of components are significantly reduced (by dealing with them in a flexible and declarative manner). Components are specifically designed to deal with unanticipated requests and they can spontaneously generate requests for assistance if they find themselves in difficulty. Moreover because these interactions are enacted through a high-level agent communication language, coupling becomes a knowledge level issue. At a stroke, this removes syntactic concerns from the types of errors caused by unexpected interactions. Secondly, the problem of managing control relationships between the software components is significantly reduced. All agents are continuously active and any coordination that is required is handled bottom-up through inter-agent interaction. Thus, the ordering of the system’s top-level goals is no longer something that has to be rigidly prescribed at design time. Rather, it becomes something that is handled in a context-sensitive manner at run-time.

From this discussion, it is apparent that a natural way to modularise a complex system is in terms of multiple, interacting, autonomous components that have particular objectives to achieve. In short, agent-oriented decompositions aid the process of developing complex systems.

3.2 Suitability of Agent-Oriented Abstractions

A significant part of the design process is finding the right models for viewing the problem. In general, there will be multiple candidates and the difficult task is picking the most appropriate one. When designing software, the most powerful abstractions are those that minimise the semantic distance between the units of analysis that are intuitively used to conceptualise the problem and the constructs present in the solution paradigm. In the case of complex systems, the problem to be characterised consists of sub-systems, sub-system components, interactions and organisational relationships. Taking each in turn:

- Sub-systems naturally correspond to agent organisations. They involve a number of constituent components that act and interact according to their role within the larger enterprise.
- The suitability of viewing sub-system components as agents has already been made (section 3.1).
- The interplay between the sub-systems and between their constituent components is most naturally viewed in terms of high level social interactions: “at any given level of abstraction, we find meaningful collections of entities that collaborate to achieve some higher level view” [Booch, 1994] pg 34. This view accords precisely with the treatment of interaction afforded by the agent-oriented approach. Agent systems are invariably described in terms of “cooperating to achieve common objectives”, “coordinating their actions” or “negotiating to resolve conflicts”.
- Complex systems involve changing webs of relationships between their various components. They also require collections of components to be treated as a single conceptual unit when viewed from a different level of abstraction. Here again the agent-oriented mindset provides suitable abstractions. A rich set of structures is typically available for explicitly representing and managing organisational relationships. Interaction protocols exist for forming new groupings and disbanding unwanted ones. Finally, structures are available for modelling collectives. The latter point is especially useful in relation to representing sub-systems since they are nothing more than a team of components working together to achieve a collective goal.

3.3 Need for Flexible Management of Changing Organisational Structures

Organisational constructs are first-class entities in agent systems. Thus explicit representations are made of organisational relationships and structures. Moreover, agent-based systems have the concomitant computational mechanisms for flexibly forming, maintaining and disbanding organisations. This representational power enables agent-oriented systems to exploit two facets of the nature of complex systems. Firstly, the notion of a primitive component can be varied according to the needs of the observer. Thus at one level, entire sub-systems can be viewed as singletons, alternatively, teams or collections of agents can be viewed as primitive

[2] Indeed the view that decompositions based upon functions/actions/processes are more intuitive and easier to produce than those based upon data/objects is even acknowledged within the object-oriented community [Meyer, 1988] pg 44.

components, and so on until the system eventually bottoms out. Secondly, such structures provide a variety of stable intermediate forms, that, as already indicated, are essential for the rapid development of complex systems. Their availability means individual agents or organisational groupings can be developed in relative isolation and then added into the system in an incremental manner. This, in turn, ensures there is a smooth growth in functionality.

3.4 Software Engineering Pragmatics

Having made the case for an agent-oriented approach to designing and building complex systems, the next step is to determine whether it will catch on as a software engineering paradigm. This question is important because the history of computing is littered with good technologies that were never widely adopted. Two key pragmatic issues are relevant here: (i) the degree to which agents represent a radical departure from current software engineering thinking and (ii) the degree to which existing software can be integrated with agents. In general, take-up is more likely if agents are consistent with the trends of software engineering (evolution rather than revolution) and if legacy software can be incorporated in a straightforward and clean manner (a brown field versus a green field scenario).

A number of trends become evident when examining the evolution of programming models from machine languages, to procedural and structured programming, to object-based and declarative programming, onto component-ware, design patterns, and software architectures. Firstly, there has been an inexorable move from languages that have their conceptual basis determined by the underlying machine architecture, to languages that have their key abstractions rooted in the problem domain. Here the agent-oriented world view is perhaps the most natural way of characterising many types of problem. Just as the real-world is populated with objects that have operations performed on them, so it is equally full of active, purposeful agents that interact to achieve their objectives[3]. Indeed, many object-oriented analyses start from precisely this perspective: “we view the world as a set of autonomous agents that collaborate to perform some higher level function” [Booch, 1994] pg. 17. Secondly, the basic building blocks of the programming models exhibit increasing degrees of localisation and encapsulation [Parunak, 1999]. Agents follow this trend by localising purpose inside each agent, by giving each agent its own thread of control, and by encapsulating action selection. Thirdly, ever richer mechanisms for promoting re-use are being provided. Here, the agent view also reaches new heights. Rather than stopping at re-use of sub-system components (design patterns and component-ware) and rigidly pre-ordained interactions (application frameworks), agents enable whole sub-systems and flexible interactions to be re-used. In the former case, agent designs and implementations are re-used within and between applications. Consider, for example, the class of agent architectures that has beliefs (what the agent knows), desires (what the agent wants) and intentions (what the agent is doing) at its core. Such Belief-Desire-Intention architectures have been used in a wide variety of applications including air traffic control, process control, fault diagnosis and transportation [Chaib-draa, 1995; Jennings, 1995; Jennings and Wooldridge, 1998]. In the latter case, flexible patterns of interaction such as the Contract Net Protocol (an agent with a task to complete advertises this fact to others who it believes are capable of performing it, these agents may submit a bid to perform the task if they are interested, and the originator then delegates the task to the agent that makes the best bid) and various forms of resource-allocation auction (e.g. English, Dutch, Vickrey) have been re-used in significant numbers of applications. In short, agent-oriented techniques represent a natural progression of current software engineering thinking and, for this reason, the main concepts and tenets of the approach should be readily acceptable to software engineering practitioners.

The second factor in favour of a rapid take up of agents is that their adoption does not require a revolution in terms of an organisation’s software capabilities. Agent-oriented systems are evolutionary and incremental as legacy (non-agent) software can be incorporated in a relatively straightforward manner [Jennings *et al.*, 1993]. The technique used is to place wrapping software around the legacy code. The wrapper presents an agent interface to the other software components and thus from the outside it looks like any other agent. On the inside, the wrapper performs a two-way translation function: taking external requests from other agents and mapping them into calls in the legacy code, and taking the legacy code’s external requests and mapping them into the appropriate set of agent communication commands. This ability to wrap legacy systems means agents may initially be

[3] Although there are some similarities between object- and agent-oriented approaches (e.g. both adhere to the principle of information hiding and recognise the importance of interactions), there are also a number of important differences. Firstly, objects are generally passive in nature: they need to be sent a message before they become active. Secondly, although objects encapsulate state and behaviour realisation they do not encapsulate behaviour activation (action choice). Thus, any object can invoke any publicly accessible method on any other object. Once the method is invoked, the corresponding actions are performed. Thirdly, object-orientation fails to provide an adequate set of concepts and mechanisms for modelling complex systems: for such systems “we find that objects, classes and modules provide an essential yet insufficient means of abstraction” [Booch, 1994] pg 34. Individual objects represent too fine a granularity of behaviour and method invocation is too primitive a mechanism for describing the types of interactions that take place. Finally, object-oriented approaches provide only minimal support for specifying and managing organisational relationships (basically relationships are defined by static inheritance hierarchies).

used as an integration technology. However, as new requirements are uncovered, so bespoke agents may be developed and added. This feature enables a complex system to grow in an evolutionary fashion (based on stable intermediate forms), while adhering to the important principle that there should always be a working version of the system available.

3.5 The Downside

Having highlighted the potential benefits of agent-oriented software engineering, this sub-section seeks to pinpoint some of the inherent difficulties associated with agent-based systems. These problems are directly attributable to the characteristics of agent-oriented software and are, therefore, intrinsic to the approach. (These complement the more pragmatic problems that are often associated with agent-oriented projects [Wooldridge and Jennings, 1998].) Naturally, since robust and reliable agent systems have been built, designers have found means of circumventing these problems. However, at this time, such solutions tend to be made on a case by case basis.

Much of the power of agents derives from the fact that they are situated problem solvers: they act in pursuit of their design objectives while maintaining an ongoing interaction with their environment. However such situatedness makes it difficult to design software capable of maintaining a balance between proactive and reactive behaviour. Leaning too much towards the former risks the agent undertaking irrelevant or infeasible tasks (as circumstances have changed). Leaning too much towards the latter means the agent may not fulfill its objectives (since it is constantly responding to short-term needs). Striking a balance requires context sensitive decision making which, in turn, means there can be a significant degree of unpredictability about which objectives the agent will pursue in which circumstances and which methods will be used to achieve the chosen objectives.

Although agent interactions represent a hitherto unseen level of sophistication and flexibility, they are also inherently unpredictable in the general case. As agents are autonomous, the patterns and the effects of their interactions are uncertain. Firstly, agents decide, for themselves at run-time, which of their objectives require interaction in a given context, which acquaintances they will interact with in order to realise these objectives, and when these interactions will occur. Hence the number, pattern and timing of interactions cannot be predicted in advance. Secondly, there is a de-coupling, and a considerable degree of variability, between what one agent first requests through an interaction and how the recipient ultimately responds. The request may be immediately honoured as it is, refused completely, or modified through some form of social interchange. In short, both the nature (a simple request versus a protracted negotiation) and the outcome of an interaction cannot be determined at the onset.

The final source of unpredictability in agent-oriented system design relates to the notion of emergent behaviour. It has long been recognised that interactive composition—collections of processes (agents) acting side-by-side and interacting in whatever way they have been designed to interact [Milner, 1993]—results in behavioural phenomena that cannot be deconstructed solely in terms of the behaviour of the individual components. This emergent behaviour is a consequence of the interaction between components. Given the sophistication and flexibility of agent interactions, it is clear that the scope for unexpected individual and group behaviour is considerable.

4 Agents for Business Process Management

This section describes an agent-based system developed for managing a British Telecom (BT) business process [Jennings *et al.*, 1996]. The particular process is providing customers with a quote for installing a network to deliver a particular type of telecommunications service. This process has a number of traits that are commonly found in corporate-wide business processes. In particular, the process is dynamic and unpredictable (it is impossible to give a complete *a priori* specification of all activities), it has a high-degree of natural concurrency, and there is a need to respect departmental and organisational boundaries.

In more detail, the following departments are involved: the customer service division (CSD), the design division (DD), the surveyor department (SD), the legal division (LD) and the various organisations that 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 an off-the-shelf item then an immediate quote can be offered. In the case of bespoke services, however, the process is more complex. CSD further analyses 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. If the requested service is legal, the design phase can start. To prepare a network design it is usually necessary to dispatch a surveyor to the customer's premises so that a detailed plan of the existing equipment can be produced. On completion of the network design and costing, DD informs CSD of the quote. CSD, in turn, informs the customer. The business process then terminates.

Following the principles of agent-oriented decomposition, the system's autonomous problem solving entities were identified (figure 2). Thus, each department is represented by an agent, as is each individual within a department. Since all these entities are active problem solvers with their own objectives, this mapping is both natural and intuitive. To achieve their individual objectives, agents need to interact with one another. In this case, all interactions take the form of negotiations about which services the agents will provide to one another and under what terms and conditions [Faratin *et al.*, 1998]. The nature of these negotiations varies depending on the context and the prevailing circumstances: negotiations between BT internal agents are more cooperative than those involving external organisations, and negotiations where time is plentiful differ from those where time is short. Thus, for example, to get a customer vetted, the CSD agent negotiates (in a competitive manner) simultaneously with all the VC agents to determine which of them can perform this service the quickest. This interaction involves generating a series of proposals and counter-proposals and if it is successful it ultimately results in a mutually agreeable contract. Generally speaking, the flexible nature of the interactions means the negotiators can tailor their behaviour to the prevailing circumstances. Thus, they can both vary the amount of utility they expect from an agreement and relax their constraints in a context dependent manner.

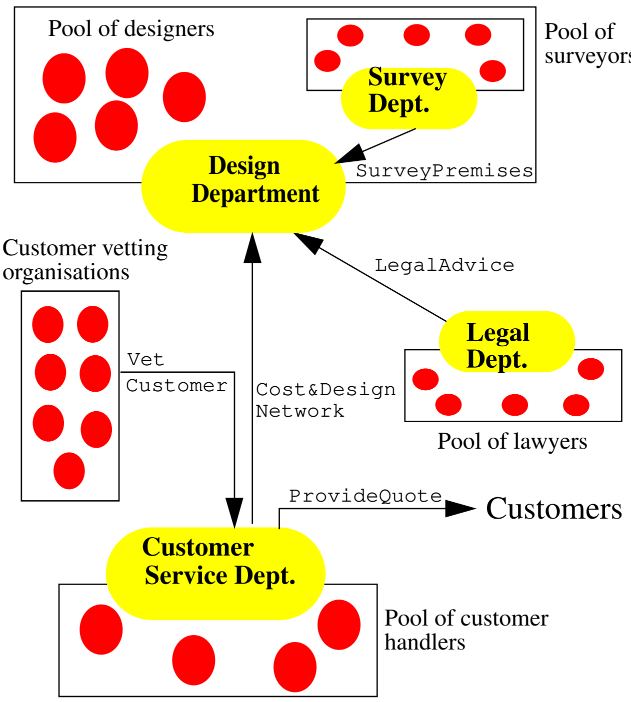

Figure 2: Agent system for managing the quote process

Direction of arrow indicates the consumer of the service labelling the arrow.

The final system design task is to characterise the organisational inter-relationships. Here, the following classes are applicable: collections of agents being grouped together as a single conceptual unit (e.g. the individual designers and lawyers in DD and LD respectively), authority relationships (e.g. the DD agent is the manager of the SD agent), peers within the same organisation (e.g. the CSD, LD, and DD agents) and customer-subcontractor relationships (e.g. the CSD agent and the various VC agents). Explicitly representing such relationships is important because it provides a means of clustering collections of agents so they can be dealt with as a single conceptual unit and because it has a significant impact on the negotiation behaviour of the participants.

5 Discussion

This paper has sought to justify the claim that agent-based computing has the potential to provide a powerful suite of metaphors, concepts and techniques for conceptualising, designing and implementing complex (distributed) systems. However, against this promise lies the perils that: (i) there is insufficient know-how about building agents that can engage in flexible social interactions; (ii) the means by which sociality impacts upon individual and collective behaviour is not well understood; and (iii) the way in which organisational relationships impact upon the behaviour of individuals and societies needs to be clarified.

One means of tackling these fundamental issues is to follow an approach that proved successful in elucidating the foundational principles and structures of individual (asocial) agents. Newell's [1982] knowledge level analysis provided the seminal characterisation of intelligent agents—it stripped away implementation and application specific details to reveal the core of asocial problem solvers. Since the aim here is to do the same for social agents, Newell's basic approach can be re-used. Thus a new computer level needs to be defined. This level can be called the Social Level [Jennings and Campos, 1997]. It should sit immediately above the knowledge level and it should provide the social principles and foundations for agent-based systems. The primary benefit of developing a social level description is that it enables the overall system's behaviour and key conceptual structures to be studied without the need to delve into the implementation details of the individual agents or the specifics of particular interaction protocols. Thus prediction of the behaviour of the social agents and of the overall system can be made more easily.

To this end, a preliminary version of the social level will be outlined (following Newell's general nomenclature):

- The *system* (the entity to be described at that level) is an agent organisation.
- The *components* of an agent organisation (the primitive elements from which it is built up) are the agents themselves and the channels through which they interact. Interactions occur because of the inherent dependencies that exist between the agents (either through the environment or as a consequence of their adopted goals) [Jennings, 1993]. Thus dependencies are also a primitive component. The final component is the organisational relationships that hold between the agents.

- *Composition laws* define how the components are assembled to form the system. In this case, the agents undertake particular roles in the organisation. These roles define the objectives of the agents and their organisational relationships, the channels through which they interact, and the patterns of their interaction. Accompanying the roles are the organisation's rules that define the laid down procedures or the emergent norms. These rules specify, among other things, who can adopt which roles and under what terms and conditions, what should happen if roles are violated, and how role conflicts should be handled.
- *Behaviour laws* determine how the system's behaviour depends upon its composition and on its components' behaviour. These laws indicate how the agents within the organisation should balance their individualistic objectives with those that stem from being part of the organisation. Here no single law is universally best; rather, there is a continuous spectrum from the purely selfish to the altruistic.
- The *medium* is the elements the system processes to obtain the behaviour it was designed to achieve. In this case, it is the social knowledge that each agent maintains about the agent organisation and its role therein. This includes, among other things, its social and organisational obligations, its mechanisms for influencing other agents and its mechanisms for altering the organisational structure.

References

[Booch, 1994] G. Booch. *Object-oriented analysis and design with applications*. Addison Wesley, 1994.

[Brooks, 1995] F. P. Brooks. *The mythical man-month*. Addison Wesley, 1995.

[Buschmann *et al.*, 1998] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad and M. Stahl. *A System of Patterns.* Wiley, 1998.

[Chaib-draa, 1995] B. Chaib-draa. Industrial applications of distributed AI. *Comms. of ACM* 38(11):47-53, 1995.

[Faratin *et al.*, 1998] P. Faratin, C. Sierra, and N. R. Jennings. Negotiation Decision Functions for Autonomous Agents. *Int. J. of Robotics and Autonomous Systems* 24(3-4):159-182, 1998.

[Gamma *et al.*, 1995] E. Gamma, R. Helm, R. Johnson and J. Vlissides. *Design Patterns*. Addison Wesley, 1995.

[Jennings, 1993] N. R. Jennings. Commitments and Conventions: The Foundation of Coordination in Multi-Agent Systems. *The Knowledge Engineering Review* 8(3):223-250, 1993.

[Jennings, 1995] N. R. Jennings. Controlling cooperative problem solving in industrial multi-agent systems. *Artificial Intelligence* 75(2): 195-240, 1995.

[Jennings and Campos, 1997] N. R. Jennings and J. R. Campos. Towards a Social Level Characterisation of Socially Responsible Agents. *IEE Proc. on Software Engineering*, 144(1):11-25, 1997.

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

[Jennings *et al.*, 1993] N. R. Jennings, L. Varga, R. P. Aarnts, J. Fuchs, and P. Skarek. Transforming stand-alone expert systems into a community of cooperating agents. *Int J. of Engineering Applications of AI* 6(4):317-331, 1993.

[Jennings and Wooldridge, 1998] N. R. Jennings and M. Wooldridge (eds.) *Agent technology: foundations, applications and markets*. Springer Verlag, 1998.

[Newell, 1982] A. Newell. The Knowledge Level. *Artificial Intelligence* 18:87-127, 1982.

[Meyer, 1988] B. Meyer. *Object-oriented software construction*. Prentice Hall, 1988.

[Milner, 1993] R. Milner. Elements of interaction. *Comms. of ACM* 36(1):78-89, 1993.

[Parunak, 1999] H. V. D. Parunak. Industrial and practical applications of DAI. In G. Weiss, editor, Multi-Agent Systems, MIT Press, 377-421, 1999.

[Russell and Norvig, 1995] S. Russell and P. Norvig. *Artificial intelligence: a modern approach*. Prentice Hall, 1995.

[Simon, 1996] H. A. Simon. *The sciences of the artificial*. MIT Press, 1996.

[Szyperski, 1998] C. Szyperski. *Component Software*. Addison Wesley, 1998.

[Wegner, 1997] P. Wegner. Why interaction is more powerful than algorithms. *Comms. of ACM* 40(5):80-91, 1997.

[Wooldridge, 1997] M. Wooldridge. Agent-based software engineering. *IEE Proc Software Engineering* 144:26-37, 1997.

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

[Wooldridge and Jennings, 1998] M. J. Wooldridge and N. R. Jennings. Pitfalls of Agent-Oriented Development. *Proc 2nd Int. Conf. on Autonomous Agents*, Minneapolis, USA, 385-391, 1998.