Agent-Oriented Software Engineering

Nicholas R. Jennings and Michael Wooldridge

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

Abstract

Agent-oriented techniques represent an exciting new means of analysing, designing and building complex software systems. They have the potential to significantly improve current practice in software engineering and to extend the range of applications that can feasibly be tackled. Yet, to date, there have been few serious attempts to cast agent systems as a software engineering paradigm. This paper seeks to rectify this omission. Specifically, it will be argued that: (i) the conceptual apparatus of agent-oriented systems is well-suited to building software solutions for complex systems and (ii) agent-oriented approaches represent a genuine advance over the current state of the art for engineering complex systems. Following on from this view, the major issues raised by adopting an agent-oriented approach to software engineering are highlighted and discussed.

1. Introduction

Designing and building high quality industrial-strength software is difficult. Indeed, it has been claimed that such development projects are among the most complex construction tasks undertaken by humans. Against this background, a wide range of software engineering paradigms have been devised (e.g., procedural programming, structured programming, declarative programming, object-oriented programming, design patterns, application frameworks and component-ware). Each successive development either claims to make the engineering process easier or to extend the complexity of applications that can feasibly be built. Although there is some evidence to support these claims, researchers continually strive for more efficient and powerful software engineering techniques, especially as solutions for ever more demanding applications are required.

This paper will argue that analysing, designing and implementing software as a collection of interacting, autonomous agents (i.e., as a *multi-agent system* [26] [39]) represents a promising point of departure for software engineering. While there is some debate about exactly what constitutes an autonomous agent and what constitutes interaction, this work seeks to abstract away from particular dogmatic standpoints. Instead, we focus on those characteristics for which there is some consensus. From this standpoint, the paper's central hypothesis will be advanced: for certain classes of problem (that will be defined), adopting a multi-agent approach to system development affords software engineers a number of significant advantages over contemporary methods. Note that we are not suggesting that multi-agent systems are a silver bullet [4]—there is no evidence to suggest they will represent an order of magnitude improvement in software engineering produc-

tivity. However, we believe that for certain classes of application, an agent-oriented approach can significantly improve the software development process.

In seeking to demonstrate the efficacy of the agent-oriented approach, the most compelling form of analysis would be to quantitatively show how adopting such techniques had improved, according to some standard set of software metrics, the development process in a range of projects. However, such data is simply not available (as it is still not for more established methods such as object-orientation). However, there are compelling arguments for believing that an agent-oriented approach will be of benefit for engineering certain complex software systems. These arguments have evolved from a decade of experience in using agent technology to construct large-scale, real-world applications in a wide variety of industrial and commercial domains [20].

The contribution of this paper is twofold. Firstly, despite multi-agent systems being touted as a technology that will have a major impact on future generation software ("pervasive in every market by the year 2000" [17] and "the new revolution in software" [14]), there has been no systematic evaluation of *why this may be the case.* Thus, although there are an increasing number of deployed agent applications (see [5], [19], [20], [27] for a review), nobody has analysed precisely what makes the paradigm so effective. This is clearly a major gap in knowledge, which this paper seeks to address. Secondly, there has been comparatively little work on viewing multi-agent systems as a software engineering. This shortcoming is rectified by recasting the essential components of agent systems into more traditional software engineering concepts, and by examining the impact on the software engineering life-cycle of adopting an agent-oriented approach.

The remainder of the paper is structured as follows. Section two makes the case for an agent-oriented approach to software engineering. It analyses the type of complexity present in industrial-strength software, characterises the key conceptual mechanisms of the agent-oriented approach, and explains how these mechanisms are well suited to tackling the complexity present in software development. Section three examines the impact of adopting an agent-oriented approach on the software engineering lifecycle—focusing in particular on the specification, implementation and verification phases. Section four deals with the pragmatics of agent-oriented software engineering by presenting some common pitfalls that frequently bedevil agent-oriented developments. Section five concludes by identifying the major open issues that need to be addressed if agent-oriented techniques are to reach the software engineering mainstream.

2. The Case for an Agent-Oriented Approach to Software Engineering

This section characterises the essential nature of real-world software systems (section 2.1) and then goes on to present exactly what we mean by the notion of agent-oriented software (section 2.2). Using these characterisations, arguments are advanced as to why agent-oriented techniques are well suited to developing complex software systems (section 2.3).

2.1 The Nature of Complex Software Systems

Industrial-strength software is complex in nature: it is typically characterised by a large number of parts that have many interactions [37]. Moreover this complexity is not accidental [4]: it is an innate property of the types of tasks for which software is used. The role of software engineering is therefore to provide structures and techniques that make it easier to handle this complexity. Fortunately, this complexity exhibits a number of important regularities [37]:

- Complexity frequently takes the form of a hierarchy[1]. That is, the system is composed of inter-related sub-systems, each of which is itself a hierarchy. The precise nature of the organisational relationships varies between sub-systems, although some generic forms (such as client-server, peer, team) can be identified. Organisational relationships are not static: they can, and frequently do, vary over time.

- The choice of which components in the system are primitive is relatively arbitrary and is defined very much 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 much more rapidly if there are stable intermediate forms, than if there are not.

- It is possible to distinguish between the interactions *among s*ub-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. Thus, 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.

Drawing these insights together, it is possible to define a canonical view of a complex system (figure 1). The system's hierarchical nature is expressed through the "composed of" links, components within a sub-system are connected through "frequent interaction" links, and interactions between components are expressed through "infrequent interaction" links. The variable notion of primitive components can be seen in the way that atomic components at one level are expanded out to entire sub-systems at subsequent levels.

[1] Here the term "hierarchy" is *not* used to mean that each sub-system is subordinated by an authority relation to the system to which it belongs. Rather, it should be interpreted in a broad sense to mean a system that is analysable into successive sets of sub-systems.

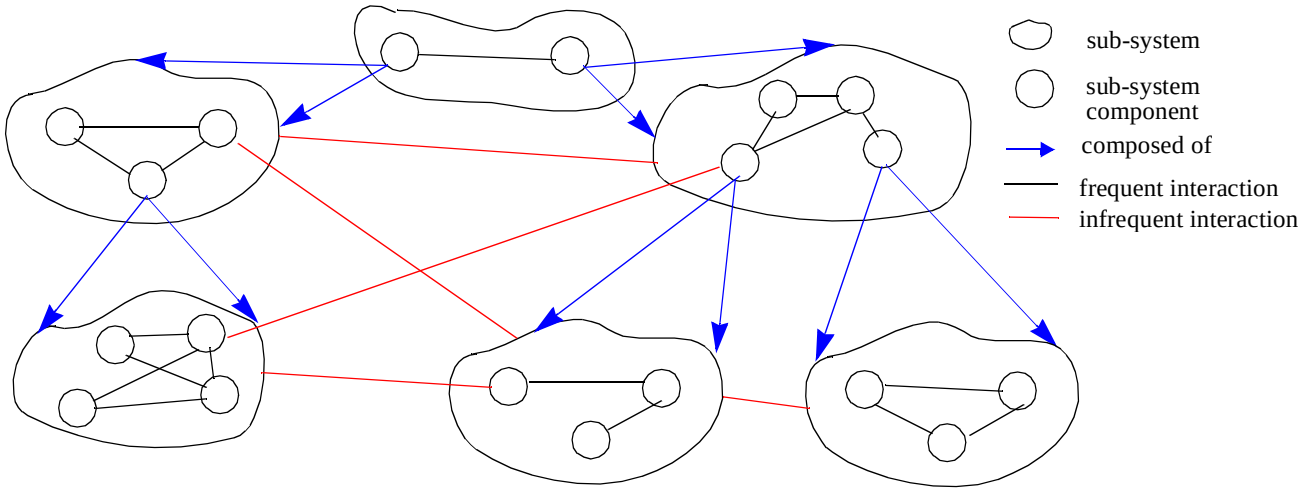

Figure 1: View of a Canonical Complex System

Given these observations, software engineers have devised a number of powerful tools in order to manage this complexity. The principal mechanisms include [3]:

- *Decomposition*: The most basic technique for tackling any large problem is to divide it into smaller, more manageable chunks each of which can then be dealt with in relative isolation (note the nearly decomposable sub-systems in figure 1). Decomposition 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*[2]: The process of identifying and managing the inter-relationships between the various problem solving components (note the sub-system and interaction links of figure 1). 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. For example, 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. For example, a number of components may need to work together in order to provide a particular functionality.

[2] Booch uses the term "hierarchy" for this final point [3]. However, hierarchy invariably gives the connotation of control, hence the more neutral term "organisation" is used here. Organisations can be arranged such that they correspond to control hierarchies, however they can also correspond to groups of peers, and anything that falls in-between.

The precise nature and way in which these tools are used varies enormously between software paradigms. Hence when characterising a new paradigm, such as agent-oriented software, clear positions need to be determined on each of these issues (section 2.2). Moreover, when assessing the power of a paradigm, arguments need to be advanced as to why the chosen way of dealing with these issues helps software engineers build systems more effectively (section 2.3).

2.2 What is Agent-Oriented Software?

At present, there is a great deal of ongoing debate about exactly what constitutes an agent, yet there is nothing approaching a universal consensus. However, an increasing number of researchers find the following characterisation useful [41]:

> *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*

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 that environment through their sensors and they act on the environment through their effectors[3]; (iii) designed to fulfil a specific role—they have particular objectives to achieve, that can either be explicitly or implicitly represented within the agents; (iv) autonomous—they have control both over their internal state and over their own behaviour; (v) capable of exhibiting flexible (context-dependent) problem solving behaviour—they need to be reactive (able to respond in a timely fashion to changes that occur in their environment in order to satisfy their design objectives) and proactive (able to opportunistically adopt new goals and take the initiative in order to satisfy their design objectives) [42].

When adopting an agent-oriented view of the world, it soon becomes apparent that a single agent is insufficient[4]. 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 else to manage the dependencies that ensue from being situated in a common environment. These interactions range from simple semantic interoperation (the ability to exchange comprehensible communications), through traditional client-server type interactions (the ability to request that a particular action is performed), to rich social interactions (the ability to cooperate, coordinate and negotiate about a course of action). Whatever the nature of the social process, however, there are two points that qualitatively differentiate agent interactions from

---

3. Typically each agent has a partial view of the environment (that may or may not overlap with that of others) and a limited sphere of influence through which it can alter the environment.

4. It can be argued that there is no such thing as a single agent system; everything involves multiple agents.

those that occur in other software engineering paradigms. Firstly, agent-oriented interactions generally occur through a high-level (declarative) agent communication language (typically based on speech act theory [1]). Consequently, interactions are usually conducted at the knowledge level [25]: 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 necessarily foreseen at design time.

In most cases, agents act to achieve objectives on behalf of individuals or companies. Thus, when agents interact there is typically some underlying organisational context. This context helps define the nature of the relationship between the agents. For example, they may be peers working together in a team, one may be the boss of the others, or they may be involved in a series of employer-subcontractor relationships. To capture such links, agent systems often have explicit constructs for modeling organisational relationships (e.g., peer, boss, etc.) and organisational structures (e.g., teams, groups, coalitions, etc.). It should be noted that in many cases, these relationships may change while the system is operating. Social interaction means existing relationships evolve (e.g., an agent awards a new contract) and new relations are created (e.g., a number of agents may form a team to deliver a particular service that no one individual can offer). The temporal extent of these relationships can vary enormously: from just long enough to deliver a particular service once to a permanent bond. To cope with this variety and dynamicity, agent researchers have expended considerable effort: devising protocols that enable organisational groupings to be formed and disbanded, specifying mechanisms to ensure groupings act together in a coherent fashion, and developing structures to characterise the macro behaviour of collectives.

Drawing these points together (figure 2), it can be seen that adopting an agent-oriented approach to software engineering means decomposing the problem into multiple, interacting, autonomous components (agents) that have particular objectives to achieve. The key abstraction models that define the “agent-oriented mindset” are agents, interactions and organisations. Finally, explicit structures and mechanisms are often available for describing and managing the complex and changing web of organisational relationships that exist between the agents.

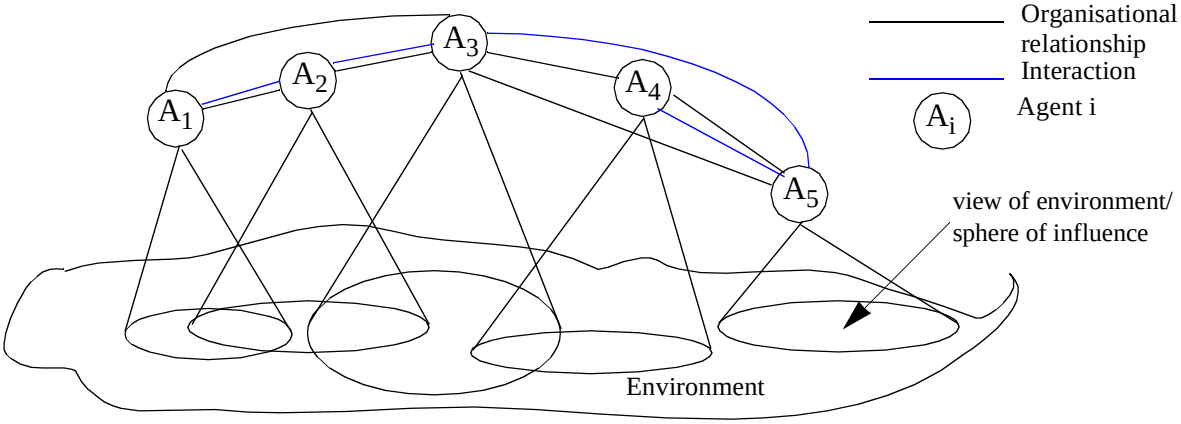

Figure 2: Canonical view of a multi-agent system

2.3 The Case for an Agent-Oriented Approach

Having characterised complex systems and described agent software, we now consider why such agent-oriented techniques are well suited to developing such software systems. This argument has three parts:

- show that agent-oriented decompositions are an effective way of partititioning the problem space of a complex system (section 2.3.1);
- show that the key abstractions of the agent-oriented mindset are a natural means of modeling complex systems (section 2.3.2);
- show that the agent-oriented philosophy for identifying and managing organisational relationships is appropriate for dealing with the dependencies and interactions that exist in a complex system (section 2.3.3).

To make the case for agent-oriented software engineering even more compelling, the final step is to argue that agent-oriented techniques represent a genuine advance over the current state of the art. To this end, the agent-oriented approach will be compared with leading-edge techniques from mainstream software engineering (section 2.3.4). In particular, this involves comparisons with object-oriented analysis and design (system is built out of interacting objects that encapsulate both data and the procedures that operate on that data [3], [23]) and with component-ware (system is built by assembling pre-existing components into some overall structure [38]).

2.3.1 Agent-Oriented Decompositions

Complex systems consist of a number of related sub-systems organised in a hierarchical fashion (figure 1). At any given level, the 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 the above situation, it is entirely natural to modularise the components in terms of the objectives they achieve[5]. In other words, each component can be thought of as achieving one or more objectives. A second important observation is that current software engineering trends are towards increasing the degrees of localisation and encapsulation in problem decompositions [27]. Applying this philosophy to objective-achieving decompositions means that the individual components should have their own thread of control (i.e., components should be active) and that they should encapsulate the information and problem solving ability needed to meet these objectives. Since the components typically have to operate in an environment in which they have only partial information, they must be able to determine, at run-time, which actions they should perform in pursuit of their objectives. In short, components need autonomy over their choice of action.

In order 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 that it is impossible to *a priori* know about all the 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 can 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. This, in turn, removes syntactic level concerns from the types of errors caused by unexpected interactions. Secondly, the problem of managing control relationships between the software components (a task that bedevils more traditional functional decompositions) is significantly reduced. All agents are continuously active and any coordination or synchronisation that is required is handled 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 can be 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 make it easier to develop complex systems.

5. 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 (see, for example, [23] pg 44).

2.3.2 The Appropriateness of the Agent-Oriented Abstractions

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

- The case for viewing sub-system components as agents has already been made above (section 2.3.1).

- 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 objects that collaborate to achieve some higher level view” [3] pg 34. This view accords precisely with the knowledge level (or even social level [18]) 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”. Thus, the agent-oriented mindset is entirely appropriate for capturing the types of interaction that occur in complex systems.

- 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. Again, this view matches closely with the abstractions provided by the agent-oriented mindset. Thus, facilities are typically provided for explicitly representing organisational relationships. Interaction protocols have been developed for forming new groupings and disbanding unwanted ones. Finally, structures are available for modeling 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.

2.3.3 The Need for Flexible Management of Changing Organisational Structures

Complex systems involve a variety of organisational relationships, ranging from peers to control hierarchies, from the short-term to the ongoing. These relationships are important for two main reasons. Firstly, they allow a number of separate components to be grouped together and treated as a single conceptual entity. Secondly, they enable the high-level links between the different entities to be characterised. Given the influence and impact of organisational relationships and structures on system behaviour, the importance of providing explicit support for flexibly specifying and managing them is self-evident. Moreover, given that these relationships frequently change, the ability to dynamically adapt to prevailing circumstances is also essential.

As already indicated, organisations are first-class entities in agent systems. Thus explicit structures and flexible mechanisms are central to the agent paradigm. This representational power, when coupled with the supporting computational mechanisms, 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 a singleton, alternatively teams or collections of agents can be viewed as primitive 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 that 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.

2.3.4 Agents versus Objects and Component-ware

There are a number of similarities between the object- and the agent- oriented views of system development. For example, both emphasise the importance of interactions between entities. However there are also a number of important differences. Firstly, objects are generally passive in nature: they need to be sent a message before they come alive. Secondly, although objects encapsulate state and behaviour, 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. In this sense, objects are obedient to one another. Whilst this approach may suffice for smaller applications in cooperative and well-controlled environments, it is not suited to either large or competitive environments because this *modus operandi* places all the onus for invoking behaviour on the client. The server has no say in the matter. Insights from both organisation and political science indicate that such one-sided approaches do not scale well. It is far better to allow the action executor to have a say in the matter (i.e., action invocation becomes a process of mutual consent). Thus, for instance, the executor, who by definition is more intimate with the details of the actions to be performed, may know of a good reason why the specified behaviour should not be invoked in the current situation. In such cases, the executor should either be able to refuse the request or at least point out the potentially damaging consequences of carrying it out. These observations become even more pertinent as software moves from the realms of being under the control of a single organisation (or an inherently cooperative group of organisations) into open environments in which the system contains organisations that compete against one another. Thirdly, object-orientation fails to provide an adequate set of concepts and mechanisms for modeling the types of system depicted in figure 1: “for complex systems we find that objects, classes and modules provide an essential yet insufficient means of abstraction” [3] pg 34. Complex systems require richer problem solving abstractions. 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. Recognition of these facts, led to the development of more powerful abstraction mechanisms such as design patterns and application frameworks [13]. Whilst these are undoubtedly a step forward, they fall short of the desiderata for complex systems developments. By their very nature, they focus on

generic system functions and the mandated patterns of interaction are rigid and pre-determined. Finally, object- oriented approaches provide minimal support for structuring collectives (basically relationships are defined by inheritance class hierarchies). Yet as illustrated in figure 1, complex systems involve a variety of organisational relationships (of which “part-of” and “is-a” are but two of the simpler kinds).

A technology that is closely related to that of object-oriented systems is *component-based software* [38]. There are a number of motivations for component-based software, but arguably the single most important driver behind component systems is the long cherished dream of software reuse. In short, industrial software has been bedevilled by the problem of “first principles” software development. By and large, most everyday software development projects develop all software components from scratch. This is obviously inefficient, as most elements of a typical software project will have been successfully implemented many times before. As a result of this, researchers have for several decades been eager to develop methods that will permit them to build software from pre-built components in the same way that electronic engineers build systems from integrated circuits. Software component architectures, of which Java beans are probably the best known example (see http://java.sun.com/beans/), are one approach to this problem. Essentially a component model allows developers to build and combine software as “units of deployment” [38]. The simplest examples of component software are from user interface design. Thus when one constructs a user interface using a language such as Java, one typically makes use of a range of pre-defined classes, to implement interface elements. Using a component architecture, these elements are available directly to the developer as single units of deployment, for direct use in an application. A developer can ask such a user interface component about the methods it supports, can customise its properties, and manipulate it entirely through a graphical development environment.

As components in the sense described here are descended from object-oriented technology, they inherit all the properties of objects, and in particular, the close relationship between objects and agents also holds true for objects and components. In addition, however, agents also share with components the concept of a *single unit of deployment.* Thus, like components, agents are typically self-contained computational entities, that do not need to be deployed along with other components in order to realise the services they provide. Also, agents are often equipped with “meta-level reasoning” ability in the sense that they are able to respond to requests for information about the services they provide. However, components are not autonomous in the way that we understand agents to be; in addition, like objects, there is no corresponding notion of reactive, proactive, or social behaviour in component software.

In making these arguments in favour of an agent-oriented approach, it is possible for proponents of object-oriented systems, component-ware, or any other programming paradigm to claim that such mechanisms can be implemented using their technique. This is undoubtedly true. Agent-oriented systems are, after all, just computer programs. However, this misses the point. The value of different software paradigms is the mindset and the techniques that they provide to software engi-

neers. In this respect, agent-oriented concepts are an extension of those available in other paradigms.

3. The Agent-Oriented Software Lifecycle

Now that we understand why agent-oriented techniques represent a promising approach for engineering complex systems, we can turn to the details of agent-oriented software engineering. In particular, we examine what specifications for agent systems might look like (section 3.1), discuss how to implement such specifications (section 3.2), and explore how to verify that implemented systems do in fact satisfy their specifications (section 3.3). See [41] for a more detailed discussion on these issues.

3.1 Specification

In this section, we consider the problem of *specifying an agent system*. What are the requirements for an agent specification framework? What sort of properties must it be capable of representing? Taking the view of agents discussed above, the predominant approach to specifying agents has involved treating them as intentional systems that may be understood by attributing to them mental states such as beliefs, desires, and intentions [9], [42]. Following this idea, a number of approaches for formally specifying agents have been developed, which are capable of representing the following aspects of an agent-based system:

- the *beliefs* that agents have—the information they have about their environment, which may be incomplete or incorrect;
- the *goals* that agents will try to achieve;
- the *actions* that agents perform and the effects of these actions;
- the *ongoing interaction* that agents have—how agents interact with each other and their environment over time.

We call a theory that explains how these aspects of agency interact to achieve the mapping from input to output an *agent theory*. The most successful approach to (formal) agent theory appears to be the use of a *temporal modal logic* (space restrictions prevent a detailed technical discussion on such logics—see, e.g., [42] for extensive references). Two of the best known such logical frameworks are the Cohen-Levesque theory of intention [8], and the Rao-Georgeff belief-desire-intention model [31]. The Cohen-Levesque model takes as primitive just two attitudes: beliefs and goals. Other attitudes (in particular, the notion of *intention*) are built up from these. In contrast, Rao-Georgeff take intentions as primitives, in addition to beliefs and goals. The key technical problem faced by agent theorists is developing a formal model that gives a good account of the interrelationships between the various attitudes that together comprise an agent's internal state

[42]. Comparatively few serious attempts have been made to specify real agent systems using such logics—see, e.g., [12] for one such attempt.

3.2 Implementation

Once given a specification, we must implement a system that is correct with respect to this specification. The next issue we consider is this move from abstract specification to concrete computational system. There are at least two possibilities for achieving this transformation that we consider here:

- somehow directly execute or animate the abstract specification; or
- somehow translate or compile the specification into a concrete computational form using an automatic translation technique.

In the sub-sections that follow, we shall investigate each of these possibilities in turn.

3.2.1 Directly Executing Agent Specifications

Suppose we are given a system specification, ϕ, which is expressed in some logical language L. One way of obtaining a concrete system from ϕ is to treat it as an *executable specification*, and *interpret* the specification directly in order to generate the agent's behaviour. Interpreting an agent specification can be viewed as a kind of constructive proof of satisfiability, whereby we show that the specification ϕ is satisfiable by *building a model* (in the logical sense) for it. If models for the specification language L can be given a computational interpretation, then model building can be viewed as executing the specification. To make this discussion concrete, consider the Concurrent MetateM programming language [11]. In this language, agents are programmed by giving them a temporal logic specification of the behaviour it is intended they should exhibit; this specification is directly executed to generate each agent's behaviour. Models for the temporal logic in which Concurrent MetateM agents are specified are linear discrete sequences of states: executing a Concurrent MetateM agent specification is thus a process of constructing such a sequence of states. Since such state sequences can be viewed as the histories traced out by programs as they execute, the temporal logic upon which Concurrent MetateM is based has a computational interpretation; the actual execution algorithm is described in [2].

Note that executing Concurrent MetateM agent specifications is possible primarily because the models upon which the Concurrent MetateM temporal logic is based are comparatively simple, with an obvious and intuitive computational interpretation. However, agent specification languages in general (e.g., the BDI formalisms of Rao and Georgeff [31]) are based on considerably more complex logics. In particular, they are usually based on a semantic framework known as *possible worlds* [6]. The technical details are somewhat involved for the purposes of this article: the main point is that, *in general*, possible worlds semantics do not have a computational interpre-

tation in the way that Concurrent MetateM semantics do. Hence it is not clear what "executing" a logic based on such semantics might mean. In response to this, a number of researchers have attempted to develop executable agent specification languages with a simplified semantic basis, that has a computational interpretation. An example is Rao's AgentSpeak(L) language, that although essentially a BDI system, has a simple computational semantics [30].

3.2.2 Compiling Agent Specifications

An alternative to direct execution is *compilation*. In this scheme, we take our abstract specification, and transform it into a concrete computational model via some automatic synthesis process. The main perceived advantages of compilation over direct execution are in run-time efficiency. Direct execution of an agent specification, as in Concurrent MetateM, above, typically involves manipulating a symbolic representation of the specification at run time. This manipulation generally corresponds to reasoning of some form, which is computationally costly. Compilation approaches aim to reduce abstract symbolic specifications to a much simpler computational model, which requires no symbolic representation. The "reasoning" work is thus done off-line, at compile-time; execution of the compiled system can then be done with little or no run-time symbolic reasoning.

Compilation approaches usually depend upon the close relationship between models for temporal/modal logic (which are typically labelled graphs of some kind), and automata-like finite state machines. For example, Pnueli and Rosner [29] synthesise reactive systems from branching temporal logic specifications. Similar techniques have also been used to develop concurrent system skeletons from temporal logic specifications. Perhaps the best-known example of this approach to agent development is the *situated automata* paradigm of Rosenschein and Kaelbling [34]. They use an epistemic logic (i.e., a logic of *knowledge* [10]) to specify the perception component of intelligent agent systems. They then used a technique based on constructive proof to directly synthesise automata from these specifications [33].

The general approach of automatic synthesis, although theoretically appealing, is limited in a number of important respects. First, as the agent specification language becomes more expressive, then even off-line reasoning becomes too expensive to carry out. Second, the systems generated in this way are not capable of *learning*, (i.e., they are not capable of adapting their "program" at run-time). Finally, as with direct execution approaches, agent specification frameworks tend to have no concrete computational interpretation, making such a synthesis impossible.

3.3 Verification

Once we have developed a concrete system, we need to show that this system is correct with respect to our original specification. This process is known as *verification*, and it is particularly important if we have introduced any informality into the development process. We can divide approaches to the verification of systems into two broad classes: (1) *axiomatic*; and (2) *semantic* (model checking). In the subsections that follow, we shall look at the way in which these two

approaches have evidenced themselves in agent-based systems.

3.3.1 Axiomatic Approaches

Axiomatic approaches to program verification were the first to enter the mainstream of computer science, with the work of Hoare in the late 1960s [16]. Axiomatic verification requires that we can take our concrete program, and from this program systematically derive a logical theory that represents the behaviour of the program. Call this the program theory. If the program theory is expressed in the same logical language as the original specification, then verification reduces to a proof problem: show that the specification is a theorem of (equivalently, is a logical consequence of) the program theory.

The development of a program theory is made feasible by *axiomatizing* the programming language in which the system is implemented. For example, Hoare logic gives us more or less an axiom for every statement type in a simple Pascal-like language. Given the axiomatization, the program theory can be derived from the program text in a systematic way.

Perhaps the most relevant work from mainstream computer science is the specification and verification of reactive systems using temporal logic, in the way pioneered by Pnueli, Manna, and colleagues [22] [28]. The idea is that the computations of reactive systems are infinite sequences, that correspond to models for linear temporal logic. Temporal logic can be used both to develop a system specification, and to axiomatize a programming language. This axiomatization can then be used to systematically derive the theory of a program from the program text. Both the specification and the program theory will then be encoded in temporal logic, and verification hence becomes a proof problem in temporal logic.

Comparatively little work has been carried out within the agent-based systems community on axiomatizing multi-agent environments. We shall review just one approach. In [40], an axiomatic approach to the verification of multi-agent systems was proposed. Essentially, the idea was to use a temporal belief logic to axiomatize the properties of two multi-agent programming languages. Given such an axiomatization, a program theory representing the properties of the system could be systematically derived in the way indicated above. A temporal belief logic was used for two reasons. First, a temporal component was required because, as we observed above, we need to capture the ongoing behaviour of a multi-agent system. A belief component was used because the agents we wish to verify are each symbolic AI systems in their own right. That is, each agent is a symbolic reasoning system, which includes a representation of its environment and desired behaviour. A belief component in the logic allows us to capture the symbolic representations present within each agent. The two multi-agent programming languages that were axiomatized in the temporal belief logic were Shoham's Agent0 [36], and Fisher's Concurrent MetateM (see above). Note that this approach relies on the operation of agents being sufficiently simple that their properties can be axiomatized in the logic. It works for Shoham's Agent0 and Fisher's Con-

current MetateM largely because these languages have a simple semantics, closely related to rule-based systems, which in turn have a simple logical semantics. For more complex agents, an axiomatization is not so straightforward. Also, capturing the semantics of concurrent execution of agents is not easy (it is, of course, an area of ongoing research in computer science generally).

3.3.2 Semantic Approaches: Model Checking

Ultimately, axiomatic verification reduces to a proof problem. Axiomatic approaches to verification are thus inherently limited by the difficulty of this proof problem. Proofs are hard enough, even in classical logic; the addition of temporal and modal connectives to a logic makes the problem considerably harder. For this reason, more efficient approaches to verification have been sought. One particularly successful approach is that of *model checking*. As the name suggests, whereas axiomatic approaches generally rely on syntactic proof, model checking approaches are based on the semantics of the specification language.

The model checking problem, in abstract, is quite simple: given a formula ϕ of language L, and a model M for L, determine whether or not ϕ is valid in M, i.e., whether or not $M \models_L \phi$. Model checking-based verification has been studied in connection with temporal logic. The technique once again relies upon the close relationship between models for temporal logic and finite-state machines. Suppose that ϕ is the specification for some system, and Π is a program that claims to implement ϕ. Then, to determine whether or not Π truly implements ϕ, we take Π, and from it generate a model M_Π that corresponds to Π, in the sense that M_Π encodes all the possible computations of Π. To determine whether or not $M_\Pi \models \phi$, i.e., whether the specification formula ϕ is valid in M_Π; the program Π satisfies the specification ϕ just in case the answer is 'yes'. The main advantage of model checking over axiomatic verification is in complexity: model checking using the branching time temporal logic CTL [7] can be done in polynomial time, whereas the proof problem for most modal logics is quite complex.

In [32], Rao and Georgeff present an algorithm for model checking agent systems. More precisely, they give an algorithm for taking a logical model for their (propositional) BDI agent specification language, and a formula of the language, and determining whether the formula is valid in the model. The technique is closely based on model checking algorithms for normal modal logics [15]. They show that despite the inclusion of three extra modalities, (for beliefs, desires, and intentions), into the CTL branching time framework, the algorithm is still quite efficient, running in polynomial time. So the second step of the two-stage model checking process described above can still be done efficiently. However, it is not clear how the first step might be realised for BDI logics. Where does the logical model characterizing an agent actually comes from—can it be derived from an arbitrary program Π, as in mainstream computer science? To do this, we would need to take a program implemented in, say, Pascal, and from it derive the belief, desire, and intention accessibility relations that are used to give a semantics to the BDI component of the logic. Because, as we noted earlier, there is no clear relationship between the BDI logic and the

concrete computational models used to implement agents, it is not clear how such a model could be derived.

4. Pitfalls of Agent-Oriented Development

Having highlighted the potential benefits of agent-oriented software engineering, this section pinpoints some of the inherent drawbacks of building software using agent technology. The following set of problems are directly attributable to the characteristics of agent-oriented software and are, therefore, intrinsic to the approach. Naturally, since robust and reliable agent systems have been built, designers have found means of circumventing these problems. However such solutions tend to be very much on a case by case basis; more general solutions are a long way off.

Much has been made of the fact that agents are situated problem solvers: they have to act in pursuit of their objectives while maintaining an ongoing interaction with their environment. 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 fulfil its objectives (since it is constantly responding to short-term needs). Striking a balance requires context-sensitive decision making. This, 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 power, 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 realize 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 decoupling, 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 is, it may be refused completely, or it may be 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 acting side-by-side and interacting in whatever way they have been designed to interact [24] — results in behavioural phenomena that cannot be generally understood 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.

As well as these specific technological problems, we can also identify a number of pitfalls that seems to occur repeatedly in agent development projects [43]

- *You oversell agent solutions, or fail to understand where agents may usefully be applied*.

  Agent technology is currently the subject of considerable attention in the computer science and AI communities, and many predictions have been made about its long term potential. However, one of the greatest current sources of perceived failure in agent development initiatives is simply the fact that developers overestimate the potential of agent systems. While agent technology represents a potentially novel and important new way of conceptualising and implementing software, it is important to understand its limitations. Agents are ultimately just software, and agent solutions are subject to the same fundamental limitations as more conventional software solutions. In particular, agent technology has *not* somehow solved the (very many) problems that have dogged AI since its inception. Agent systems typically make use of AI techniques [35]. In this sense, they are an application of AI technology. But their "intelligent" capabilities are limited by the state of the art in this field. Artificial intelligence as a field has suffered considerably from over-optimistic claims about its potential. Most recently, perhaps, the expert systems experience vividly illustrates the perils of overselling a promising technology. It seems essential to us that agent technology does not fall prey to this same problem. Realistic, sober expectations of what agent technology can provide are thus extremely important.

- *You get religious or dogmatic about agents*

  Although agents have been used in a wide range of applications, they are not a universal solution. There are many applications for which conventional software development paradigms (such as object-oriented programming) are more appropriate. Indeed, given the relative immaturity of agent technology and the small number of deployed agent applications, there should be clear advantages to an agent based solution before such an approach is even contemplated.

- *You don't know why you want agents*

  This is a common problem for any new technology that has been hyped as much as agents. Managers read optimistic financial forecasts of the potential for agent technology and, not surprisingly, they want part of this revenue. However in many cases, managers that propose an agent project do not actually have a clear idea about what "having agents" will buy them. In short, they have no business model for agents – they have no understanding of how agents can be used to enhance their existing products, how they can enable them to generate new product lines, and so on.

- *You want to build generic solutions to one-off problems*

  This is a pitfall to which many software projects fall victim, but it seems to be especially prevalent in the agent community. It typically manifests itself in the devising of an archi-

tecture or testbed that supposedly enables a whole range of potential types of system to be built, when what is really required is a bespoke design to tackle a single problem. In such situations, a custom built solution will be easier to develop and far more likely to satisfy the requirements of the application.

- *You believe that agents are a silver bullet*

  The holy grail of software engineering is a "silver bullet": a technique that will provide an order of magnitude improvement in software development. Agent technology is a newly emerged, and as yet essentially untested software paradigm: it is only a matter of time before someone claims agents are a silver bullet. This would be dangerously naive. As we pointed out above, there are good arguments in favour of the view that agent technology will lead to improvements in the development of complex distributed software systems. But, as yet, these arguments are largely untested in practice.

- *You forget you are developing software*

  At the time of writing, the development of any agent system – however trivial – is essentially a process of experimentation. There are no tried and trusted techniques available to assist the developer in producing agent software which has a guaranteed performance profile. Unfortunately, because the process is experimental, it encourages developers to forget that they are actually developing software. The result is a foregone conclusion: the project flounders, not because of agent-specific problems, but because basic software engineering good practice was ignored.

- *You forget you are developing multi-threaded software*

  Multi-threaded systems have long been recognised as one of the most complex classes of computer system to design and implement. By their very nature, multi-agent systems tend to be multi-threaded (both within an agent and certainly within the society of agents). So, in building a multi-agent system, it is vital not to ignore the lessons learned from the concurrent and distributed systems community – the problems inherent in multi-threaded systems do not go away, just because you adopt an agent-based approach.

- *Your design doesn't exploit concurrency*

  One of the most obvious features of a poor multi-agent design is that the amount of concurrent problem solving is comparatively small or even in extreme cases non-existent. If there is only ever a need for a single thread of control in a system, then the appropriateness of an agent-based solution must seriously be questioned.

- *You decide you want your own agent architecture*

  Agent architectures are essentially designs for building agents [42]. When first attempting an agent project, there is a great temptation to imagine that no existing agent architecture

meets the requirements of your problem, and that it is therefore necessary to design one from first principles. But designing an agent architecture from scratch in this way is often a mistake: our recommendation is therefore to study the various architectures described in the literature [42], and either license one or else implement an "off the shelf" design.

- *Your agents use too much AI*

  When one builds an agent application, there is an understandable temptation to focus exclusively on the agent-specific, "intelligence" aspects of the application. The result is often an agent framework that is too overburdened with experimental techniques (natural language interfaces, planners, theorem provers, reason maintenance systems, …) to be usable. In general, a more successful short term strategy is to build agents with a minimum of AI techniques.

- *You see agents everywhere*

  When one learns about multi-agent systems for the first time, there is a tendency to view *everything* as an agent. This is perceived to be in some way conceptually pure. But if one adopts this viewpoint, then one ends up with agents for everything – including agents for addition and subtraction. It is not difficult to see that naively viewing everything as an agent in this way will be extremely inefficient: the overheads of managing agents and inter-agent communication will rapidly outweigh the benefits of an agent-based solution. Moreover, we do not believe it is useful to refer to very fine-grained computational entities as agents [42].

- *You have too few agents*

  While some designers imagine a separate agent for every possible task, others appear not to recognise the value of a multi-agent approach at all. They create a system that completely fails to exploit the power offered by the agent paradigm, and develop a solution with a very small number of agents doing all the work. Such solutions tend to fail the standard software engineering test of coherence, which requires that a software module should have a single, coherent function. The result is rather as if one were to write an object-oriented program by bundling all the functionality into a single class. It can be done, but the result is not pretty.

- *You spend all your time implementing infrastructure*

  One of the greatest obstacles to the wider use of agent technology is that there are no widely-used software platforms for developing multi-agent systems. Such platforms would provide all the basic infrastructure (for message handling, tracing and monitoring, run-time management, and so on) required to create a multi-agent system. As a result, almost every multi-agent system project that we have come across has had a significant portion of available resources devoted to implementing this infrastructure from scratch. During this implementation stage, valuable time (and hence money) is often spent imple-

menting libraries and software tools that, in the end, do little more than exchange messages across a network. By the time these libraries and tools have been implemented, there is frequently little time, energy, or enthusiasm left to work either on the agents themselves, or on the cooperative/social aspects of the system.

- *Your agents interact too freely or in an unorganised way.*

  The dynamics of multi-agent systems are complex, and can be chaotic. Often, the only way to find out what is likely to happen is to run the system repeatedly. If a system contains many agents, then the dynamics can become too complex to manage effectively. Another common misconception is that agent-based systems require no real structure. While this may be true in certain cases, most agent systems require considerably more system-level engineering than this. Some way of structuring the society is typically needed to reduce the system's complexity, to increase the system's efficiency, and to more accurately model the problem being tackled.

In this section, we hope to have highlighted some of the main pitfalls that await the agent system developer. Note that our intention has emphatically *not* been to indicate (unintentionally or otherwise) that agent-based development is any more complex or error-prone than more traditional software engineering approaches. Rather, we recognise that there are certain pitfalls which seem common to agent-based solutions – just as there are certain pitfalls which seem common to object-oriented solutions. By recognising these pitfalls, we cannot guarantee the success of an agent-based development project, but we can at least eliminate some of the more obvious sources of failure.

5. Conclusions

In this article, we have described why we perceive agents to be a significant technology for software engineering. We have discussed in detail how the characteristics of certain complex systems appear to indicate the appropriateness of an agent-based solution: as with objects before them, agents represent a natural *abstraction* mechanism with which to decompose and organise complex systems. In addition, we have summarised some of the key issues in the specification, implementation, and verification of agent-based systems, and drawn parallels with similar work from more mainstream computer science. In particular, we have shown how many of the formalisms and techniques developed for specifying, implementing, and verifying agent systems are closely related to those developed for what are known as *reactive* systems in mainstream computing. Finally, we have described some of the pitfalls of agent-based development.

Software engineering for agent systems is at an early stage of development, and yet the widespread acceptance of the concept of an agent implies that agents have a significant future in software engineering. If the technology is to be a success, then its software engineering aspects will need to be taken seriously. Probably the most important outstanding issues for agent-based software engineering are: (i) an understanding of the situations in which agent solutions are appropriate; and (ii) principled but *informal* development techniques for agent systems. While some

attention has been given to the latter (in the form of analysis and design methodologies for agent systems [21]), almost no attention has been given to the former (but see [43]).

References

[1] J. L. Austin (1962) “How to do things with words” Clarendon Press, Oxford.

[2] H. Barringer, M. Fisher, D. Gabbay, G. Gough, and R. Owens (1989) “Concurrent MetateM: A framework for programming in temporal logic” REX Workshop on Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness (LNCS Volume 430), 94-129. Springer-Verlag.

[3] G. Booch (1994) “Object-oriented analysis and design with applications” Addison Wesley.

[4] F. P. Brooks (1995) “The mythical man-month” Addison Wesley.

[5] B. Chaib-draa (1995) “Industrial applications of distributed AI” Comms. of ACM 38 (11) 47-53.

[6] B. Chellas (1980) “Modal Logic: An Introduction” Cambridge University Press.

[7] E. M. Clarke and E. A. Emerson (1981) “Design and synthesis of synchronization skeletons using branching time temporal logic” In D. Kozen, editor, Logics of Programs (LNCS Volume 131), 52-71, Springer-Verlag.

[8] P. R. Cohen and H. J. Levesque (1990) “Intention is choice with commitment” Artificial Intelligence, 42 213-261.

[9] D. C. Dennett (1987) “The Intentional Stance” The MIT Press.

[10] R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi (1995) “Reasoning About Knowledge” The MIT Press.

[11] M. Fisher (1997) “An alternative approach to concurrent theorem proving” In J. Geller, H. Kitano, and C. B. Suttner, editors, Parallel Processing in Artificial Intelligence 3, 209-230. Elsevier Science.

[12] M. Fisher and M. Wooldridge (1997) “On the formal specification and verification of multi-agent systems” Int. Journal of Cooperative Information Systems 6 (1) 37-65.

[13] E. Gamma, R. Helm, R. Johnson and J. Vlissides (1995) “Design patterns: elements of reusable object-oriented software” Addison Wesley.

[14] C. Guilfoyle and E. Warner (1994) “Intelligent agents: the new revolution in software” Ovum.

[15] J. Y. Halpern and M. Y. Vardi (1991) “Model checking versus theorem proving: A manifesto” In V. Lifschitz, editor, AI and Mathematical Theory of Computation—Papers in Honor of John McCarthy, 151-176, Academic Press.

[16] C. A. R. Hoare (1969) “An axiomatic basis for computer programming” Comms. of the ACM, 12 (10) 576-583.

[17] P. C. Janca (1995) “Pragmatic application of information agents” BIS Strategic Report.

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

[19] N. R. Jennings, K. Sycara and M. Wooldridge (1998) “A Roadmap of Agent Research and Development” Int Journal of Autonomous Agents and Multi-Agent Systems 1 (1) 7-38.

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

[21] D. Kinny and M. Georgeff (1997) “Modelling and design of multi-agent systems” In J. P. Mueller, M. Wooldridge, and N. R. Jennings, editors, Intelligent Agents III (LNAI Volume 1193), 1-20. Springer-Verlag.

[22] Z. Manna and A. Pnueli (1995) “Temporal Verification of Reactive Systems—Safety” Springer-Verlag.

[23] B. Meyer (1988) “Object-oriented software construction” Prentice Hall.

[24] R. Milner (1993) “Elements of interaction” Comms. of ACM 36 (1) 78-89.

[25] A. Newell, (1993) “Reflections on the Knowledge Level” *Artificial Intelligence* **59** 31-38.

[26] G. M. P. O’Hare and N. R. Jennings (editors) (1996) “Foundations of distributed artificial intelligence” John Wiley & Sons.

[27] H. V. D. Parunak (1999) “Industrial and practical applications of DAI” In G. Weiss, editor, Multi-Agent Systems, MIT Press.

[28] A. Pnueli (1986) “Specification and development of reactive systems” In Information Processing 86, Elsevier Science Publishers.

[29] A. Pnueli and R. Rosner (1989) “On the synthesis of a reactive module” In Proceedings of the 16th ACM Sym. on the Principles of Programming Languages, 179-190.

[30] A. S. Rao (1996) “AgentSpeak(L): BDI agents speak out in a logical computable language” In W. Van de Velde and J. W. Perram, editors, Agents Breaking Away: Proceedings of the 7th

European Workshop on Modelling Autonomous Agents in a Multi-Agent World, (LNAI Volume 1038), 42-55. Springer-Verlag.

[31] A. S. Rao and M. Georgeff (1995) “BDI Agents: from theory to practice” Proc. of the 1st Int. Conf. on Multi-Agent Systems, 312-319, San Francisco, CA.

[32] A. S. Rao and M. P. Georgeff (1993) “A model-theoretic approach to the verification of situated reasoning systems” Proc. of the 13th Int. Joint Conf. on Artificial Intelligence, 318-324, Chambery, France.

[33] S. Rosenschein and L. P. Kaelbling (1986) “The synthesis of digital machines with provable epistemic properties” In J. Y. Halpern, editor, Proc. of the 1986 Conf. on Theoretical Aspects of Reasoning About Knowledge, 83-98. Morgan Kaufmann.

[34] S. J. Rosenschein and L. P. Kaelbling (1996) “A situated view of representation and control” In P. E. Agre and S. J. Rosenschein, editors, Computational Theories of Interaction and Agency, 515-540. The MIT Press.

[35] S. Russell and P. Norvig (1995) “Artificial Intelligence: A Modern Approach” Prentice-Hall.

[36] Y. Shoham (1993) “Agent-oriented programming” Artificial Intelligence, 60 (1) 51-92.

[37] H. A. Simon (1996) “The sciences of the artificial” MIT Press.

[38] C. A. Szyperski (1997) “Component software: beyond object-oriented programming” Addison Wesley.

[39] G. Weiss (1999) “Multi-agent systems” MIT Press.

[40] M. Wooldridge (1992) “The Logical Modelling of Computational Multi-Agent Systems” Ph.D. thesis, Department of Computation, UMIST, Manchester, UK.

[41] M. Wooldridge (1997) “Agent-based software engineering” IEE Proc. on Software Engineering, 144 (1) 26-37.

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

[43] M. Wooldridge and N. R. Jennings (1998) “Pitfalls of agent-oriented development” Proceedings of the 2nd Int. Conf. on Autonomous Agents, 385-391, Minneapolis/St. Paul, MN.