Agent-Based Software Engineering

Michael Wooldridge

Mitsubishi Electric Digital Library Group
18th Floor, Centre Point, 103 New Oxford Street
London WC1A 1EB, United Kingdom

mjw@dlib.com

September 19, 1997

Abstract

The technology of intelligent agents and multi-agent systems seems set to radically alter the way in which complex, distributed, open systems are conceptualized and implemented. The purpose of this paper is to consider the problem of building a multi-agent system as a software engineering enterprise. The article focuses on three issues: (i) how agents might be specified; (ii) how these specifications might be refined or otherwise transformed into efficient implementations; and (iii) how implemented agents and multi-agent systems might subsequently be verified, in order to show that they are correct with respect to their specifications. These issues are discussed with reference to a number of case-studies. The article concludes by setting out some issues and open problems for future research.

1 Introduction

> *Intelligent agents are ninety-nine percent computer science and one percent AI.*
> Oren Etzioni [12]

Over its 40-year history, Artificial Intelligence (AI) has been subject to many and varied criticisms. Perhaps the most persistent and troubling of these is that AI has simply *failed to deliver* on its promises. Clearly, the more extreme predictions of some AI researchers (such as human-quality intelligent robots within five decades) have not been realized. This would not be so worrying if it was obvious that AI had paid off in some other way: if, for example, AI techniques were standard components in workaday software. But this is not the case. Even comparatively mundane AI techniques (such as rule-based systems) are still regarded as homeopathic medicine by a significant proportion of the mainstream computer science community. Why is this? There are many reasons, including, for example, the reluctance of software developers to learn about and apply new technologies, and the inappropriateness of mainstream software engineering techniques and tools for AI system development. But at least part of the answer is also that many AI researchers either ignore or else gloss over the pragmatic concerns of software development, for the simple reason that they do not regard themselves as software engineers. AI prides itself on being multi-disciplinary, taking contributions from many other fields; but software engineering is generally regarded as neither a contributor nor a concern.

The most recent infants to emerge from the AI nursery are the notions of an *intelligent agent* and *agent-based system* [66]. An intelligent agent is generally regarded as an autonomous decision making system, which senses and acts in some environment (we discuss the question of what an agent is in more detail below). Agents appear to be a promising approach to developing many complex applications, ranging from INTERNET-based electronic commerce and information gathering to industrial process control (see [28] for a survey). But unless researchers recognise that agent-based systems are about computer science and software engineering *more* than they are about AI, then within a decade, we may well be asking why agent technology suffered the same fate as so many other AI ideas that seemed good in principle.

In summary, the aim of this paper is to consider the problem of building agent-based systems as a software engineering enterprise. In so doing, the paper constructs a framework within which future work on agent-based software engineering may be placed. The paper begins by motivating and introducing the idea of agent-based systems, and then goes on to discuss the key software engineering issues of specification, refinement/implementation, and verification with respect to agent-based systems. We begin by briefly discussing the question of what a specification is, and go on to consider what an *agent-based* specification might look like. We then discuss some of the dimensions along which an agent-based specification framework might vary, with particular reference to the notion of agents as rational, mentalistic systems [57, 49]. We subsequently discuss the key issue of implementing or refining agent-based specifications into systems, and finally, we consider the verification of agent-based systems. Throughout the article, we take care both to illustrate the various issues with case studies, and to draw parallels with more mainstream software engineering research wherever possible. The article concludes with a discussion of future work directions.

It should be noted that the emphasis of this paper is on *formal* methods for agent-based software engineering. This bias reflects the current state of the field. As the area matures, and more agent-based systems are deployed, we will naturally see an increasing number of structured but informal development techniques emerge.

2 Agent-Based Systems

By an *agent-based system*, we mean one in which the key abstraction used is that of an *agent*. Agent-based systems may contain a single agent, (as in the case of user interface agents or software secretaries [39]), but arguably the greatest potential lies in the application of *multi*-agent systems [6]. By an *agent*, we mean a system that enjoys the following properties [66, pp116–118]:

- *autonomy*: agents encapsulate some state (that is not accessible to other agents), and make decisions about what to do based on this state, without the direct intervention of humans or others;
- *reactivity*: agents are *situated* in an environment, (which may be the physical world, a user via a graphical user interface, a collection of other agents, the INTERNET, or perhaps many of these combined), are able to *perceive* this environment (through the use of potentially imperfect sensors), and are able to respond in a timely fashion to changes that occur in it;

- *pro-activeness*: agents do not simply act in response to their environment, they are able to exhibit goal-directed behaviour by *taking the initiative*;

- *social ability*: agents interact with other agents (and possibly humans) via some kind of *agent-communication language* [17], and typically have the ability to engage in social activities (such as cooperative problem solving or negotiation) in order to achieve their goals.

To more clearly understand what is meant by these properties, consider the following scenario. Imagine an autonomous automatic pilot controlling an aircraft, that we present with the goal of safely landing at some airport. We expect the system to plan how to achieve this goal (perhaps by making use of pre-compiled plans, rather than reasoning from first-principles), and if necessary, we expect it to generate subsidiary goals (e.g., ascend to an altitude of 30,000 feet, then proceed due north at a speed of…). This is what we mean by pro-activeness. We also expect the system to try to execute its plans, but not blindly. Thus, in the event of unforeseen circumstances (e.g., a change in weather conditions, a fault in the aircraft, a request from air-traffic control), we expect the system to respond to the new situation accordingly, in time for the response to be useful. A system that spent hours deliberating about what to do next would be no use as an auto-pilot. This is what we mean by reactiveness. We also expect our auto-pilot to be able to cooperate with air-traffic controllers and perhaps other aircraft in order to achieve its goals. This is what we mean by social ability. Note that this example is intended to be extreme: it is not a claim about what agents can currently do, or what they will be able to do in the near future. It is merely intended to highlight the type of features we ultimately hope to see in agents.

The concept of an intelligent autonomous agent did not appear in a vacuum. It is a natural development of various other trends in AI and computer science. In the subsections that follow, we discuss some ancestors of agents, and identify the attributes that make them distinct from their forbears.

Agents and AI

The discipline of intelligent agents has emerged largely from research in AI. In fact, one way of defining AI is as the problem of building an intelligent agent (Rusell and Norvig's recent textbook on AI more-or-less takes this view [53]). But it is important to distinguish between the broad intelligence that is the ultimate goal of the AI community, and the intelligence we seek in agents. The only intelligence requirement we generally make of our agents is that they can make an acceptable decision about what action to perform next in their environment, in time for this decision to be useful. Other requirements for intelligence will be determined by the domain in which the agent is applied: not all agents will need to be capable of learning, for example. Capabilities such as commonsense reasoning (*à la* CYC [19]) are not required for many important application domains. Thus, as we pointed out in the introduction, we view the application and exploitation of agent technology primarily as a computer science problem. Agents are simply software components that must be designed and implemented in much the same way that other software components are. However, AI techniques *are* often the most appropriate way of building agents.

Agents and Expert Systems

Expert systems were *the* AI technology of the 1980s [23]. An expert system is one that is capable of solving problems or giving advice in some knowledge-rich domain [26]. A classic example of an expert system is MYCIN, which was intended to assist physicians in the treatment of blood infections in humans. Perhaps the most important distinction between agents and expert systems is that expert systems like MYCIN are inherently *disembodied*. By this, we mean that they do not interact directly with any environment: they get their information not via sensors, but through a user acting as middle man. In addition, expert systems are not usually required to operate in anything like real-time. Finally, we do not generally require expert systems to be capable of co-operating with other agents.

Despite these differences, some expert systems, (particularly those that perform real-time control tasks), look very much like agents. A good example is ARCHON [27], which started life as a collection of expert systems, and ended up being viewed as a multi-agent system. ARCHON operates in the domain of industrial process control.

Agents and Objects

Object-oriented development techniques have been promoted as 'best practice' by the academic computer science community for at least a decade. Fueled by recent popular interest in languages such as JAVA, object-oriented approaches are finally leaving the relative backwater of academia and entering the mainstream. While there is much ongoing debate about many aspects of object-oriented development, there is broad agreement that an object is an entity that encapsulates some state and a collection of methods, corresponding to operations that may be performed on that state. Methods are typically invoked as a result of messages sent to the object (one may think of these messages as requests for services).

The most obvious difference between the 'standard' object model and our view of agent-based systems is that in traditional object-oriented programs, there is a single thread of control. In contrast, agents are process-like, concurrently executing entities. However, there have been variants on the basic object model in which objects are more like processes: object-based concurrent programming models such as ACTORS [1] have long been recognized as an elegant model for concurrent computation, and 'active object' systems are also quite similar; even comparatively early on in the development of object-oriented programming, it was recognized that something like agents would be a natural next step.

Given these remarks, it may seem that agents are identical to (active) objects in most important respects: they encapsulate both state and behaviour, and communicate via message passing[1]. But agents are *not* simply objects by another name. This is because an agent is a *rational decision making system*: we require an agent to be capable of reactive and pro-active behaviour, and of interleaving these types of behaviour as the situation demands. The object-oriented research community has nothing whatsoever to say about building systems that are capable of this kind of behaviour. In contrast, the design of such systems is a fundamental research topic

[1] The following is from a textbook on object-oriented programming:

> 'There is a tendency [. . .] to think of objects as "actors" and endow them with human-like intentions and abilities. It's tempting to think about objects "deciding" what to do about a situation, [and] "asking" other objects for information. [. . .] Objects are not passive containers for state and behaviour, but are said to be the agents of a program's activity." [44, p7]

in the intelligent agents community [66][2]. In addition, the object-oriented community has not addressed issues like cooperation, competition, negotiation, computational economies, and so on, that form the foundation for multi-agent systems development [6].

2.1 Agents as Rational Systems

An obvious problem is how to conceptualize systems that are capable of rational behaviour of the type discussed above. One of the most successful solutions to this problem involves viewing agents as *intentional systems* [10], whose behaviour can be predicted and explained in terms of *attitudes* such as belief, desire, and intention [47]. The rationale for this approach is that in everyday life, we use a *folk psychology* to explain and predict the behaviour of complex intelligent systems: people. For example, we use statements such as *Michael intends to write a paper* in order to explain Michael's behaviour. Once told this statement, we expect to find Michael shelving other commitments and developing a plan to write the paper; we would expect him to spend a lot of time at his computer; we would not be surprised to find him in a grumpy mood; but we *would* be surprised to find him at a late night party.

This *intentional stance*, whereby the behaviour of a complex system is understood via the attribution of attitudes such as believing and desiring, is simply an *abstraction tool*. It is a convenient shorthand for talking about complex systems, which allows us to succinctly predict and explain their behaviour without having to understand how they actually work. Now, much of computer science is concerned with looking for good abstraction mechanisms, since these allow system developers to manage complexity with greater ease: witness procedural abstraction, abstract data types, and most recently, objects. So, why not use the intentional stance as an abstraction tool in computing — to explain, understand, and, crucially, *program* complex computer systems?

For many researchers in AI, this idea of programming computer systems in terms of 'mentalistic' notions such as belief, desire, and intention is the key component of agent-based computing. The concept was articulated most clearly by Yoav Shoham, in his *agent-oriented programming* (AOP) proposal [57]. There seem to be a number of arguments in favour of AOP. First, it offers us a familiar, non-technical way to talk about complex systems. We need no formal training to understand mentalistic talk: it is part of our everyday linguistic ability.

Secondly, AOP may be regarded as a kind of 'post-declarative' programming. In procedural programming, saying what a system should do involves stating precisely *how* to do it, by writing a detailed algorithm. Procedural programming is difficult because it is hard for people to think in terms of the detail required. In declarative programming (*à la* PROLOG), the aim is to reduce the emphasis on control aspects: we state a goal that we want the system to achieve, and let a built-in control mechanism (e.g., goal-directed refutation theorem proving) figure out what to do in order to achieve it. However, in order to successfully write efficient or large programs in a language like PROLOG, it is necessary for the programmer to have a detailed understanding of how the built-in control mechanism works. This conflicts with one of the main goals of declarative programming: to relieve the user of the need to deal with control issues. In AOP, the idea is that, as in declarative programming, we state our goals, and let the built-in control mechanism figure out what to do in order to achieve them. In this case, however, the control mechanism implements some model of rational agency (such as the Cohen-Levesque theory of intention [9], or the Rao-Georgeff BDI model [47]). Hopefully, this computational

[2] Of course, both the idea of an agent and the idea of an object both owe a great debt to the pioneering work of Hewitt on open systems and the ACTOR model [1].

model corresponds to our own intuitive understanding of (say) beliefs and desires, and so we need no special training to use it. Returning to the example we used above, suppose we told our autopilot agent to land at LAX airport. Then we would expect the agent to continue to attempt to land at LAX until it had succeeded, or else it discovered that this goal was impossible. We would not expect the agent to accept any other goal that was not consistent with landing at LAX, and we would expect the agent to continue attempting different strategies for achieving the goal in the event of difficulties. Ideally, as AOP programmers, we would not be concerned with *how* the agent achieves its goals. The reality, as ever, does not quite live up to the ideal.

Interestingly, we again find that researchers from a more mainstream computing discipline have adopted a similar set of ideas. In theoretical computer science, *logics of knowledge* are used to reason about distributed systems [13]. The idea is that when specifying such systems, one often makes use of statements such as the following:

$$\begin{array}{ll}\text{IF} & \text{process } i \textit{ knows} \text{ process } j \text{ has received message } m_1 \\ \text{THEN} & \text{process } i \text{ should send process } j \text{ message } m_2.\end{array} \qquad (1)$$

This kind of statement may be formalised using a logic of knowledge. Although most work on knowledge theory in distributed systems has been rather abstract, and unconnected with software practice, researchers have recently begun to pay serious attention to the possibility of directly programming systems using statements such as (1), above [13, pp233–271]. The idea is very close to AOP.

Now that we understand what an agent is, we can begin to look at software engineering for agent-based systems.

3 Specification

The software development process begins by establishing the client's requirements. When this process is complete, a *specification* is developed, which sets out the functionality of the new system. The purpose of this section is to consider what a specification for an agent-based system might look like. What are the requirements for an agent specification framework? What sort of properties must it be capable of representing? To answer this question, we return to the properties of agents, as discussed in the preceding section.

We observed above that agents are situated in an environment, and are able to perceive this environment through sensors of some kind. Agents thus have *information* about their environment. This leads to our first requirement: that the agent specification framework must be capable of representing both the state of the environment itself, and the information an agent has about the environment. It is worth making some comments about what properties this information might have. First, the information an agent has may be incorrect. The agent's sensors may be faulty, the information might be out of date, or the agent may have been deliberately or accidentally given false information. Secondly, the information an agent has is not directly available to other agents: agents do not share data structures, and do not have access to the private data structures of other agents (this is part of what we meant by autonomy). Third, the environment will contain other agents, each with their own information about the environment. Thus an agent may have information about the state of other agents: we may need to represent such 'nested' information. Note that it is common practice to refer to the information available to an agent as that agent's *beliefs*.

Now consider the notion of *reactivity*. Software systems may be broadly divided into two types: *functional* and *reactive* [45]. A functional system is one that simply takes some input, performs some computation over this input, and eventually produces some output. Such systems may be viewed as functions $f : I \to O$ from a set I of inputs to O of outputs. The classic example of such a system is a compiler, which can be viewed as a mapping from a set I of legal source programs to a set O of corresponding object or machine code programs. Although the internal complexity of the mapping may be great (e.g., in the case of a *really* complex programming language), it is nevertheless the case that functional programs are, in general, inherently simpler to specify, design, and implement than reactive systems. Because functional systems terminate, it is possible to use pre- and post-condition formalisms in order to reason about them [24]. In contrast, reactive systems do not terminate, but rather maintain an *ongoing interaction* with their environment. It is therefore not possible to use pre- and post-condition formalisms such as Hoare logic to reason about them. Instead, reactive systems must be specified in terms of their *ongoing behaviour*. The next requirement for our agent specification framework is that it must be capable of representing this inherently reactive nature of agents and multi-agent systems. Note that one of the most successful formalisms developed for specifying reactive systems is temporal logic. The idea is that when specifying a reactive system, one often wants to state requirements such as 'if a request is received, then a response is eventually sent'. Such requirements are easily and elegantly expressed in temporal logic.

The third aspect of agents as discussed above is *pro-activeness*, by which we mean that agents are able to exhibit goal-directed behaviour. (Note that we use the term 'goal' fairly loosely. We include such notions as commitments or obligations in our usage.) It does not follow that in order to exhibit goal-directed behaviour, an agent must explicitly generate and represent goals [38], although this is by far the most common approach. Our agent specification framework must be capable of representing these *conative* (goal-directed) aspects of agency.

Finally, our agents are able to *act*. Agents do not typically have *complete* control over their environment (our auto-pilot cannot control the weather), but they are generally able to *influence* their environment by performing actions, and may have reliable control over portions of it. We require some way of representing these actions within our specification framework. To summarize, an agent specification framework must be capable of capturing at least the following aspects of an agent-based system:

- the *beliefs* agents have;
- the *ongoing interaction* agents have with their environment;
- the *goals* that agent will try to achieve;
- the *actions* that agents perform and the effects of these actions.

What sort of specification framework is capable of representing such aspects of a system? The most successful approach appears to be the use of a *temporal modal logic* [7] (space restrictions prevent a detailed technical discussion on such logics — see, e.g., [66] for a detailed overview and extensive references). A typical temporal modal agent specification framework will contain:

- normal modal logic connectives for representing agent's beliefs;
- temporal logic connectives for representing the dynamics of the system — its ongoing behaviour;

Information Aspects
- knowledge [13]
- belief [33]
- collective information attitudes [20]

Temporal Aspects
- linear *vs* branching [11]
- dense *vs* discrete [4]
- direct reference *vs* tense operators [56]
- point based *vs* interval based [2]

Conative Aspects
- desire
- intention [9]
- obligation [42]
- choice [5]
- collective pro-attitudes [34]

Actions
- direct representation [22]
- implicit representation [55]

Table 1: Dimensions of Agent Specification

- normal modal logic connectives for representing conatives (e.g., desires, intentions, obligations);
- some apparatus for representing the actions that agents perform.

Given these requirements, there are a great many dimensions along which an agent specification framework may vary: some of these dimensions are summarized in Table 1. Note that there is by no means any consensus on the desirable properties of what we might call an 'agent theory'. For example, two of the best known agent theories are the Cohen-Levesque theory of intention [9], and the Rao-Georgeff belief-desire-intention model [47]. 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. Also, Cohen-Levesque adopt a linear temporal model, (and state a number of objections to branching temporal models), whereas branching time is more or less essential to the Rao-Georgeff model. As a result, the two formalisms, though closely related, are irreconcilable.

The key technical problem faced by agent theorists is developing a model that gives a good account of the interrelationships between the various attitudes that together comprise an agents internal state. Some contemporary models are reviewed in [66].

Case Study: The Belief-Desire-Intention Model

One of the most successful agent theories is the *belief-desire-intention* (BDI) model of Rao and Georgeff (see [47] for extensive references). The technical details of BDI are somewhat involved, and so here, we shall simply summarize the main concepts that underpin BDI models.

As the name suggests, the internal state of a BDI agent is comprised of three key data structures, which are intended to loosely correspond to beliefs, desires, and intentions. An agent's *beliefs* are intended to represent the information it has about the world, as we suggested above. Beliefs will typically be represented symbolically: in the Procedural Reasoning System (PRS) — the best-known BDI implementation — beliefs look very much like PROLOG facts [18]. An agent's *desires* may be thought of as the tasks allocated to it. An agent may not in fact be able to achieve all of its desires, and in humans, desires may even be inconsistent. An agent's *intentions* represent desires that it has committed to achieving. The intuition is that as agents will not, in general, be able to achieve *all* their desires, even if these desires *are* consistent, they must therefore fix upon some subset of available desires and commit resources to achieving them. Chosen desires are *intentions*. These intentions will then feedback into future decision making: for example, an agent should not in future adopt intentions that conflict with those it currently holds.

The BDI model of agency has been formalised by Rao and Georgeff in a family of *BDI logics* [48, 51]. These logics are extensions to the expressive branching time logic CTL* [11], which also contain normal modal connectives for representing beliefs, desires, and intentions. Most work on BDI logics has focussed on possible relationships between the three 'mental states' [48], and more recently, on developing proof methods for restricted forms of the logics [51].

3.1 Discussion

Specification languages for agent-based systems are an order of magnitude more complex than the comparatively simple temporal and modal languages that have become commonplace in mainstream computer science. Typically, they are temporal logics enriched by a family of additional modal connectives, for representing the 'mental state' of an agent. There are a number of problems with such languages, in addition their conceptual complexity. The most worrying of these is with respect to their semantics. While the temporal component of these logics tends to be rather standard, the semantics for the additional modal connectives are given in the normal modal logic tradition of *possible worlds* [7]. So, for example, an agent's beliefs in some state are characterised by a set of different states, each of which represents one possibility for how the world could actually be, given the information available to the agent. In much the same way, an agents desires in some state are characterised by a set of states that are consistent with the agents desires. Intentions are represented similarly. There are several advantages to the possible worlds model: it is well-studied and well-understood, and the associated mathematics (known as *correspondence theory*) is extremely elegant. These attractive features make possible worlds the semantics of choice for almost every researcher in formal agent theory. However, there are also a number of serious drawbacks to possible worlds semantics. First, possible worlds semantics imply that agents are logically perfect reasoners, (in that their deductive capabilities are sound and complete), and they have infinite resources available for reasoning. No real agent, artificial or otherwise, has these properties.

Secondly, possible worlds semantics are generally *ungrounded*. That is, there is usually no precise relationship between the abstract accessibility relations that are used to characterize an agents state, and any concrete computational model. As we shall see in later sections, this makes it is difficult to go from a formal specification of a system in terms of beliefs, desires, and so on, to a concrete computational system. Similarly, given a concrete computational system, there is generally no way to determine what the beliefs, desires, and intentions of that system are. If temporal modal logics of the type discussed above are to be taken seriously as *specification* languages, then this problem is significant.

4 Implementation

Specification is not the end of the story in software development. 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 model. There are at least three possibilities for achieving this transformation:

1. manually refine the specification into an executable form via some principled but informal refinement process (as is the norm in most current software development);
2. somehow directly execute or animate the abstract specification; or
3. 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.

4.1 Refinement

At the time of writing, most software developers use structured but informal techniques to transform specifications into concrete implementations. Probably the most common techniques in widespread use are based on the idea of top-down refinement. In this approach, an abstract system specification is *refined* into a number of smaller, less abstract sub-system specifications, which together satisfy the original specification. If these sub-systems are still too abstract to be implemented directly, then they are also refined. The process recurses until the derived sub-systems are simple enough to be directly implemented. Throughout, we are obliged to demonstrate that each step represents a true refinement of the more abstract specification that preceded it. This demonstration may take the form of a formal proof, if our specification is presented in, say, Z [60] or VDM [29]. More usually, justification is by informal argument.

For functional systems, the refinement process is well understood, and comparatively straightforward. Refinement calculi exist, which enable the system developer to take a pre- and post-condition specification, and from it systematically derive an implementation through the use of proof rules [43]. Part of this reason for this comparative simplicity is that there is often an easily understandable relationship between the pre- and post-conditions that characterize an operation and the program structures required to implement it.

For reactive systems, refinement is not so straightforward. This is because reactive systems must be specified in terms of their ongoing behaviour. In contrast to pre- and post-condition formalisms, it is not so easy to determine what program structures are required to realise such specifications. The refinement problem for agent-based systems, where specifications may be regarded as even more abstract than those for reactive systems, is harder still. As a consequence, researchers have only just begun to investigate the refinement of agent-based systems. In the subsections that follow, we shall review two examples of this work.

Case Study: Agents in Z

Luck and d'Inverno have developed an agent specification framework in the Z language, although, as we shall see, the types of agents considered in this framework are somewhat different from those discussed above [36]. They define a four-tiered hierarchy of the entities that can exist in an agent-based system. They start with *entities*, which are inanimate objects — they have attributes (colour, weight, position), but nothing else. They then define *objects* to be entities that have capabilities (e.g., tables are entities that are capable of supporting things). *Agents* are then defined to be objects that have goals, and are thus in some sense active; finally, *autonomous agents* are defined to be agents with motivations. The idea is that a chair could be viewed as taking on my goal of supporting me when I am using it, and can hence be viewed as an agent for me. But we would not view a chair as an *autonomous* agent, since it has no motivations (and cannot easily be attributed them). Starting from this basic framework, Luck and d'Inverno go on to examine the various relationships that might exist between agents of different types. In [37], they examine how an agent-based system specified in their framework might be implemented. They found that there was a natural relationship between their hierarchical agent specification framework and object-oriented systems:

> 'The formal definitions of agents and autonomous agents rely on inheriting the properties of lower-level components. In the Z notation, this is achieved through schema inclusion [. . .]. This is easily modelled in ¡C++ by deriving one class from another. [. . .] Thus we move from a principled but abstract theoretical framework through a more detailed, yet still formal, model of the system, down to an object-oriented implementation, preserving the hierarchical structure at each stage.' [37]

The Luck-d'Inverno formalism is attractive, particularly in the way that it captures the relationships that can exist between agents. The emphasis is placed on the notion of agents acting for another, rather than on agents as rational systems, as we discussed above. The types of agents that the approach allows us to develop are thus inherently different from the 'rational' agents discussed above. So, for example, the approach does not help us to construct agents that can interleave pro-active and reactive behaviour. This is largely a result of the chosen specification language: Z. This language is inherently geared towards the specification of operation-based, functional systems. The basic language has no mechanisms that allow us to easily specify the ongoing behaviour of an agent-based system[3].

Case Study: A Methodology for BDI Agents

In section 3, we noted that the belief-desire-intention (BDI) model is one of the most successful general frameworks for agency. In [32], Kinny *et al* propose a four-stage design methodology for systems of BDI agents. The methodology is closely linked to a specific realization of the BDI model: the PRS architecture [18]. The methodology may be summarized as follows:

1. Identify the relevant *roles* in the application domain, and on the basis of these, develop an *agent class hierarchy*. An example role might be weather monitor, whereby agent *i* is required to make agent *j* aware of the prevailing weather conditions every hour.

[3] There are of course extensions to Z designed for this purpose.

2. Identify the responsibilities associated with each role, the services required by and provided by the role, and then determine the *goals* associated with each service. With respect to the above example, the goals would be to find out the current weather, and to make agent *j* aware of this information.

3. For each goal, determine the plans that may be used to achieve it, and the context conditions under which each plan is appropriate. With respect to the above example, a plan for the goal of making agent *j* aware of the weather conditions might involve sending a message to *j*.

4. Determine the belief structure of the system — the information requirements for each plan and goal. With respect to the above example, we might propose a unary predicate $windspeed(x)$ to represent the fact that the current wind speed is x. A plan to determine the current weather conditions would need to be able to represent this information.

Note that the analysis process will be iterative, as in more traditional methodologies. The outcome will be a model that closely corresponds to the PRS agent architecture. As a result, the move from end-design to implementation using PRS is relatively simple.

Kinny *et al* illustrate their methodology by applying it to an implemented air traffic management system called OASIS. This system, currently being deployed at Sidney airport in Australia, is, by any measure, a large and difficult application. It is arguably the most significant agent application yet developed. That the agent approach has been successfully applied in this domain is encouraging; the use of the methodology even more so.

4.2 Directly Executing Agent Specifications

One major disadvantage with manual refinement methods is that they introduce the possibility of error. If no proofs are provided, to demonstrate that each refinement step is indeed a true refinement, then the correctness of the implementation process depends upon little more than the intuitions of the developer. This is clearly an undesirable state of affairs for applications in which correctness is a major issue. One possible way of circumventing this problem, that has been widely investigated in mainstream computer science, is to get rid of the refinement process altogether, and *directly execute* the specification.

It might seem that suggesting the direct execution of complex agent specification languages is naive. (It is exactly the kind of suggestion that detractors of symbolic AI hate.) One should be therefore be very careful about what claims or proposals one makes. However, in certain circumstances, the direct execution of agent specification languages *is* possible.

What does it mean, to execute a formula φ of logic L? It means generating a logical model, M, for φ, such that $M \models \varphi$ [15]. If this could done without interference from the environment — if the agent had complete control over its environment — then execution would reduce to constructive theorem proving, where we show that φ is satisfiable by building a model for φ. In reality of course, agents are *not* interference-free: they must iteratively construct a model in the presence of input from the environment. Execution can then be seen as a two-way iterative process:

- environment makes something true;

- agent responds by doing something, i.e., making something else true in the model;

- environment responds, making something else true;
- ...

Execution of logical languages and theorem proving are thus closely related. This tells us that the execution of sufficiently rich (quantified) languages is not possible (since any language equal in expressive power to first-order logic is undecidable).

A useful way to think about execution is as if the agent is *playing a game* against the environment. The specification represents the goal of the game: the agent must keep the goal satisfied, while the environment tries to prevent the agent doing so. The game is played by agent and environment taking it in turns to build a little more of the model. If the specification ever becomes false in the (partial) model, then the agent loses. In real reactive systems, the game is never over: the agent must continue to play forever. Of course, some specifications (logically inconsistent ones) cannot ever be satisfied. A *winning strategy* for building models from (satisfiable) agent specifications in the presence of arbitrary input from the environment is an execution algorithm for the logic.

Case Study: Concurrent METATEM

Concurrent METATEM is a programming language for multi-agent systems, that is based on the idea of directly executing linear time temporal logic agent specifications [16, 14]. A Concurrent METATEM system contains a number of concurrently executing agents, each of which is programmed by giving it a temporal logic specification of the behaviour it is intended the agent should exhibit. An agent specification has the form $\bigwedge_i P_i \Rightarrow F_i$, where P_i is a temporal logic formula referring only to the present or past, and F_i is a temporal logic formula referring to the present or future. The $P_i \Rightarrow F_i$ formulae are known as *rules*. The basic idea for executing such a specification may be summed up in the following slogan:

on the basis of the past *do* the future.

Thus each rule is continually matched against an internal, recorded *history*, and if a match is found, then the rule *fires*. If a rule fires, then any variables in the future time part are instantiated, and the future time part then becomes a *commitment* that the agent will subsequently attempt to satisfy. Satisfying a commitment typically means making some predicate true within the agent. Here is a simple example of a Concurrent METATEM agent definition:

$$
\begin{array}{l}
\odot ask(x) \Rightarrow \Diamond give(x) \\
(\neg ask(x)\ \mathcal{Z}\ (give(x) \wedge \neg ask(x)) \Rightarrow \neg give(x) \\
give(x) \wedge give(y) \Rightarrow (x = y)
\end{array}
$$

The agent in this example is a controller for a resource that is infinitely renewable, but which may only be possessed by one agent at any given time. The controller must therefore enforce mutual exclusion. The predicate $ask(x)$ means that agent x has asked for the resource. The predicate $give(x)$ means that the resource controller has given the resource to agent x. The resource controller is assumed to be the only agent able to ‘give’ the resource. However, many agents may ask for the resource simultaneously. The three rules that define this agent’s behaviour may be summarized as follows:

Rule 1: if someone asks, then eventually give;

Rule 2: don't give unless someone has asked since you last gave; and

Rule 3: if you give to two people, then they must be the same person (i.e., don't give to more than one person at a time).

Note that Concurrent METATEM agents can communicate by asynchronous broadcast message passing, though the details are not important here.

4.3 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 (and in many cases, simply impracticable for systems that must operate in anything like real time). In contrast, 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. As a result, execution is much faster. The advantages of compilation over direct execution are thus those of compilation over interpretation in mainstream programming.

Compilation approaches usually depend upon the close relationship between models for temporal/modal logic (which are typically labeled graphs of some kind), and automata-like finite state machines. Crudely, the idea is to take a specification φ, and do a *constructive proof* of the implementability of φ, wherein we show that the specification is satisfiable by systematically attempting to build a model for it. If the construction process succeeds, then the specification is satisfiable, and we have a model to prove it. Otherwise, the specification is unsatisfiable. If we have a model, then we 'read off' the automaton that implements φ from its corresponding model. The most common approach to constructive proof is the *semantic tableaux* method of Smullyan [59].

In mainstream computer science, the compilation approach to automatic program synthesis has been investigated by a number of researchers. Perhaps the closest to our view is the work of Pnueli and Rosner [46] on the automatic synthesis of reactive systems from branching time temporal logic specifications. The goal of their work is to generate reactive systems, which share many of the properties of our agents (the main difference being that reactive systems are not generally required to be capable of rational decision making in the way we described above). To do this, they specify a reactive system in terms of a first-order branching time temporal logic formula $\forall x \; \exists y \; A \; \varphi(x, y)$: the predicate φ characterises the relationship between inputs to the system (x) and outputs (y). Inputs may be thought of as sequences of environment states, and outputs as corresponding sequences of actions. The A is a branching time temporal logic connective meaning 'on all paths', or 'in all possible futures'. The specification is intended to express the fact that in all possible futures, the desired relationship φ holds between the inputs to the system, x, and its outputs, y. The synthesis process itself is rather complex: it involves generating a Rabin tree automaton, and then checking this automaton for emptiness. Pnueli and Rosner show that the time complexity of the synthesis process is double exponential in the size of the specification, i.e., $O(2^{2^{c.n}})$, where c is a constant and $n = |\varphi|$ is the size of the specification φ. The size of the synthesized program (the number of states it contains) is of the same complexity.

The Pnueli-Rosner technique is rather similar to (and in fact depends upon) techniques developed by Wolper, Vardi, and colleagues for synthesizing Büchi automata from linear temporal logic specifications [61]. Büchi automata are those that can recognise *ω-regular expressions*: regular expressions that may contain infinite repetition. A standard result in temporal logic theory is that a formula φ of linear time temporal logic is satisfiable if and only if there exists a Büchi automaton that accepts just the sequences that satisfy φ. Intuitively, this is because the sequences over which linear time temporal logic is interpreted can be viewed as ω-regular expressions. This result yields a decision procedure for linear time temporal logic: to determine whether a formula φ is satisfiable, construct an automaton that accepts just the (infinite) sequences that correspond to models of φ; if the set of such sequences is empty, then φ is unsatisfiable. The technique for constructing an automaton from the corresponding formula is closely based on Wolper's tableau proof method for temporal logic [62].

Similar automatic synthesis techniques have also been deployed to develop concurrent system skeletons from temporal logic specifications. Manna and Wolper present an algorithm that takes as input a linear time temporal logic specification of the *synchronization* part of a concurrent system, and generates as output a CSP program skeleton ([25]) that realizes the specification [41]. The idea is that the functionality of a concurrent system can generally be divided into two parts: a functional part, which actually performs the required computation in the program, and a synchronization part, which ensures that the system components cooperate in the correct way. For example, the synchronization part will be responsible for any mutual exclusion that is required. The synthesis algorithm, (like the synthesis algorithm for Büchi automata, above), is based on Wolper's tableau proof method for temporal logic [62]. Very similar work is reported by Clarke and Emerson [8]: they synthesize synchronization skeletons from branching time temporal logic (CTL) specifications.

Case Study: Situated Automata

Perhaps the best-known example of this approach to agent development is the *situated automata* paradigm of Rosenschein and Kaelbling [52, 31]. In this approach, an agent has two main components:

- a *perception* part, which is responsible for observing the environment, and updating the internal state of the agent; and

- an *action* part, which is responsible for deciding what action to perform, based on the internal state of the agent.

Rosenschein and Kaelbling developed two programs to support the development of the perception and action components of an agent respectively. The RULER program takes a declarative perception specification and compiles it down to a finite state machine. The specification is given in terms of a theory of knowledge. The semantics of knowledge in the declarative specification language are given in terms of possible worlds, in the way described above. Crucially, however, the possible worlds underlying this logic are given a precise computational interpretation, in terms of the states of a finite state machine. It is this precise relationship that permits the synthesis process to take place.

The action part of an agent in Rosenschein and Kaelbling's framework is specified in terms of *goal reduction rules*, which encode information about how to achieve goals. The GAPPS program takes as input a goal specification, and a set of goal reduction rules, and generates as output a set of *situation action rules*, which may be thought of as a lookup table, defining what the agent should do under various circumstances, in order to achieve the goal. The process of deciding what to do is then very simple in computational terms, involving no reasoning at all. (A similar technique, called *universal plans*, was developed by Schoppers [54].)

4.4 Discussion

Structured but informal refinement techniques are the mainstay of real-world software engineering. If agent-oriented techniques are ever to become widely used outside the academic community, then informal, structured methods for agent-based development will be essential. One possibility for such techniques, followed by Luck and d'Inverno, is to use a standard specification technique (in their case, Z), and use traditional refinement methods (in their case, object-oriented development) to transform the specification into an implementation. This approach has the advantage of being familiar to a much larger user-base than entirely new techniques, but suffers from the disadvantage of presenting the user with no features that make it particularly well-suited to agent specification. It seems certain that there will be much more work on manual refinement techniques for agent-based systems in the immediate future, but exactly what form these techniques will take is not clear.

Now consider the possibility of directly executing agent specifications. A number of problems immediately suggest themselves. The first is that of finding a concrete computational interpretation for the agent specification language in question. To see what we mean by this, consider models for the agent specification language in Concurrent METATEM. These are very simple: essentially just linear discrete sequences of states. Temporal logic is (amongst other things) simply a language for expressing *constraints* that must hold between successive states. Execution in Concurrent METATEM is thus a process of generating constraints as past-time antecedents are satisfied, and then trying to build a next state that satisfies these constraints. Constraints are expressed in temporal logic, which implies that they may only be in certain, regular forms. Because of this, it is possible to devise an algorithm that is guaranteed to build a next state if it is possible to do so. Such an algorithm is described in [3].

The agent specification language upon which Concurrent METATEM is based thus has a concrete computational model, and a comparatively simple execution algorithm. Contrast this state of affairs with the kinds of temporal modal agent specification languages discussed in section 3, where we have not only a temporal dimension to the logic, but also modalities for referring to beliefs, desires, and so on. In general, these models have *ungrounded* semantics. That is, the semantic structures that underpin these logics (typically accessibility relations for each of the modal operators) have no concrete computational interpretation. As a result, it is not clear how such agent specification languages might be executed.

Another obvious problem is that execution techniques based on theorem proving are inherently limited when applied to sufficiently expressive (first-order) languages, as first-order logic is undecidable. However, complexity is a problem even in the propositional case. For 'vanilla' propositional logic, the decision problem for satisfiability is NP-complete [13, p72]; for linear temporal logic, the problem is PSPACE-complete [58]; for simple modal logics of knowledge, the problem is NP-complete, and for more complex modal logics of knowledge, the problem is EXPTIME-complete [13, p73]; for logics that *combine* temporal and (S5) modal aspects, the decision problem varies from PSPACE-complete in the simplest case to Π_1^1-complete, (and hence *undecidable*) in the propositional case, depending on what semantic assumptions are made [13, p289].

Turning to automatic synthesis, we find that the techniques described above have been developed primarily for propositional specification languages. If we attempt to extend these techniques to more expressive, first-order specification languages, then we again find ourselves coming up against the undecidability of quantified logic. Even in the propositional case, the theoretical complexity of theorem proving for modal and temporal logics is likely to limit the effectiveness of compilation techniques: given an agent specification of size 1000, a synthesis algorithm that runs in exponential time when used off-line is no more useful than an execution algorithm which runs in exponential time on-line.

Another problem with respect to synthesis techniques is that they typically result in finite-state, automata like machines, that are strictly less powerful than Turing machines. In particular, the systems generated by the processes outlined above cannot modify their behaviour at run-time. In short, they cannot learn. While for many applications, this is acceptable — even desirable — for equally many others, it is not. In expert assistant agents, of the type described in [39], learning is pretty much the *raison d'etre*. Attempts to address this issue are described in [30].

5 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. For example, any manual refinement, done without a formal proof of refinement correctness, creates the possibility of a faulty transformation from specification to implementation. Verification is the process of convincing ourselves that the transformation was sound. 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.

5.1 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 [24]. 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. Once 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 [40]. The idea is that the computations of reactive systems are infinite sequences, which correspond to models for linear temporal logic[4]. 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.

Case Study: Axiomatizing two Multi-Agent Languages

In [63], 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 [57], and Fisher's Concurrent METATEM(see above). The basic approach was as follows:

1. First, a simple abstract model was developed of symbolic AI agents. This model captures the fact that agents are symbolic reasoning systems, capable of communication. The model gives an account of how agents might change state, and what a computation of such a system might look like.

2. The histories traced out in the execution of such a system were used as the semantic basis for a temporal belief logic. This logic allows us to express properties of agents modelled at stage (1).

3. The temporal belief logic was used to axiomatize the properties of a multi-agent programming language. This axiomatization was then used to develop the program theory of a multi-agent system.

4. The proof theory of the temporal belief logic was used to verify properties of the system [65].

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 Concurrent 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,

[4] The set of all computations of a reactive system is a tree-like structure, corresponding to a model for branching time temporal logic [11].

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).

5.2 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 $M \models_L \varphi$. Model checking-based verification has been studied in connection with temporal logic [35]. 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 proceed as follows:

- 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;
- determine whether or not $M_\pi \models \varphi$, 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 ([8]) can be done in polynomial time ($O(|\varphi| \times |M|)$, where $|\varphi|$ is the size of the formula to be checked, and $|M|$ is the size of the model against which φ is to be checked — the number of states it contains)[5].

Case Study: Model Checking BDI Systems

In [50], Rao and Georgeff present an algorithm for model checking AOP 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 [21]. 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

[5] Perhaps surprisingly, model checking for linear temporal logic is actually more complex than for branching time (PSPACE-complete).

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.

5.3 Discussion

Axiomatic approaches to the verification of multi-agent systems suffer from two main problems. First, the temporal verification of reactive systems relies upon a simple model of concurrency, where the actions that programs perform are assumed to be atomic. We cannot make this assumption when we move from programs to agents. The actions we think of agents as performing will generally be much more coarse grained. As a result, we need a more realistic model of concurrency. One possibility, investigated in [64], is to model agent execution cycles as intervals over the real numbers, in the style of the temporal logic of reals [4]. The second problem is the difficulty of the proof problem for agent specification languages. As we noted in section 3, the theoretical complexity of proof for many of these logics is quite daunting.

With respect to model-checking approaches, the main problem, as we indicated above, is again the issue of ungrounded semantics for agent specification languages. If we cannot take an arbitrary program and say, for this program, what its beliefs, desires, and intentions are, then it is not clear how we might verify that this program satisfied a specification expressed in terms of such constructs.

6 Conclusions

Agent-based systems are a promising development, not just for AI, but for computer science generally. If intelligent agent technology succeeds, then it will provide a solution to many important but difficult software problems. The challenge now before the intelligent agent community is to ensure that the techniques developed particularly over the past decade for building rational agents make a smooth transition from the research lab to the desk of the everyday computer worker. This is by no means easy, as the expert systems experience demonstrates. If the community is to succeed in this endeavour, then it will need to take very seriously the comment by Oren Etzioni, that opened this paper: agents are more a problem of computer science and software engineering than AI.

In this paper, we have set out a roadmap for work in agent-based software engineering. We have examined the fundamental problems of specification, implementation, and verification from the point of view of agent-based systems. Throughout, we have been careful to draw as many parallels as possible with more mainstream software engineering.

Acknowledgments

I would like to thank Michael Fisher for the (many) discussions we have had on software engineering for agent-based systems, and also Adam Kellett and Nick Jennings for their comments on this paper. This work was supported by the EPSRC under grant GR/K57282.

References

[1] G. Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems*. The MIT Press: Cambridge, MA, 1986.

[2] J. F. Allen. Towards a general theory of action and time. *Artificial Intelligence*, 23(2):123–154, 1984.

[3] H. Barringer, M. Fisher, D. Gabbay, G. Gough, and R. Owens. METATEM: A framework for programming in temporal logic. In *REX Workshop on Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness (LNCS Volume 430)*, pages 94–129. Springer-Verlag: Heidelberg, Germany, June 1989.

[4] H. Barringer, R. Kuiper, and A. Pnueli. A really abstract concurrent model and its temporal logic. In *Proceedings of the Thirteenth ACM Symposium on the Principles of Programming Languages*, pages 173–183, 1986.

[5] N. Belnap and M. Perloff. Seeing to it that: a canonical form for agentives. *Theoria*, 54:175–199, 1988.

[6] A. H. Bond and L. Gasser, editors. *Readings in Distributed Artificial Intelligence*. Morgan Kaufmann Publishers: San Mateo, CA, 1988.

[7] B. Chellas. *Modal Logic: An Introduction*. Cambridge University Press: Cambridge, England, 1980.

[8] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In D. Kozen, editor, *Logics of Programs — Proceedings 1981 (LNCS Volume 131)*, pages 52–71. Springer-Verlag: Heidelberg, Germany, 1981.

[9] P. R. Cohen and H. J. Levesque. Intention is choice with commitment. *Artificial Intelligence*, 42:213–261, 1990.

[10] D. C. Dennett. *The Intentional Stance*. The MIT Press: Cambridge, MA, 1987.

[11] E. A. Emerson and J. Y. Halpern. 'Sometimes' and 'not never' revisited: on branching time versus linear time temporal logic. *Journal of the ACM*, 33(1):151–178, 1986.

[12] O. Etzioni. Moving up the information food chain: Deploying softbots on the world-wide web. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, Portland, OR, 1996.

[13] R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. *Reasoning About Knowledge*. The MIT Press: Cambridge, MA, 1995.

[14] M. Fisher. A survey of Concurrent METATEM — the language and its applications. In D. M. Gabbay and H. J. Ohlbach, editors, *Temporal Logic — Proceedings of the First International Conference (LNAI Volume 827)*, pages 480–505. Springer-Verlag: Heidelberg, Germany, July 1994.

[15] M. Fisher. Executable temporal logic. *The Knowledge Engineering Review*, 1996.

[16] M. Fisher and M. Wooldridge. Executable temporal logic for distributed A.I. In *Proceedings of the Twelfth International Workshop on Distributed Artificial Intelligence (IWDAI-93)*, pages 131–142, Hidden Valley, PA, May 1993.

[17] M. R. Genesereth and S. P. Ketchpel. Software agents. *Communications of the ACM*, 37(7):48–53, July 1994.

[18] M. P. Georgeff and A. L. Lansky. Reactive reasoning and planning. In *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87)*, pages 677–682, Seattle, WA, 1987.

[19] R. V. Guha and D. B. Lenat. Enabling agents to work together. *Communications of the ACM*, 37(7):127–142, July 1994.

[20] J. Y. Halpern. Knowledge and common knowledge in a distributed environment. *Journal of the ACM*, 37(3), 1990.

[21] J. Y. Halpern and M. Y. Vardi. Model checking versus theorem proving: A manifesto. In V. Lifschitz, editor, *AI and Mathematical Theory of Computation — Papers in Honor of John McCarthy*. Academic Press, 1991.

[22] D. Harel. Dynamic logic. In D. Gabbay and F. Guenther, editors, *Handbook of Philosophical Logic Volume II — Extensions of Classical Logic*, pages 497–604. D. Reidel Publishing Company: Dordrecht, The Netherlands, 1984. (Synthese library Volume 164).

[23] F. Hayes-Roth, D. A. Waterman, and D. B. Lenat, editors. *Building Expert Systems*. Addison-Wesley: Reading, MA, 1983.

[24] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, 1969.

[25] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21:666–677, 1978.

[26] P. Jackson. *Introduction to Expert Systems*. Addison-Wesley: Reading, MA, 1986.

[27] N. R. Jennings, J. Corera, I. Laresgoiti, E. H. Mamdani, F. Perriolat, P. Skarek, and L. Z. Varga. Using ARCHON to develop real-world DAI applications for electricity transportation management and particle accelerator control. *IEEE Expert*, dec 1996.

[28] N. R. Jennings and M. Wooldridge. Applying agent technology. *Applied Artificial Intelligence*, 9(6):357–370, 1995.

[29] C. B. Jones. *Systematic Software Development using VDM (second edition)*. Prentice Hall, 1990.

[30] L. P. Kaelbling. *Learning in Embedded Systems*. The MIT Press: Cambridge, MA, 1993.

[31] L. P. Kaelbling and S. J. Rosenschein. Action and planning in embedded agents. In P. Maes, editor, *Designing Autonomous Agents*, pages 35–48. The MIT Press: Cambridge, MA, 1990.

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

[33] K. Konolige. *A Deduction Model of Belief*. Pitman Publishing: London and Morgan Kaufmann: San Mateo, CA, 1986.

[34] H. J. Levesque, P. R. Cohen, and J. H. T. Nunes. On acting together. In *Proceedings of the Eighth National Conference on Artificial Intelligence (AAAI-90)*, pages 94–99, Boston, MA, 1990.

[35] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proceedings of the Eleventh ACM Symposium on the Principles of Programming Languages*, pages 97–107, 1984.

[36] M. Luck and M. d'Inverno. A formal framework for agency and autonomy. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95)*, pages 254–260, San Francisco, CA, June 1995.

[37] M. Luck, N. Griffiths, and M. d'Inverno. From agent theory to agent constrution: A case study. In *Intelligent Agents III — Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages (ATAL-96)*. Springer-Verlag: Heidelberg, Germany, 1997.

[38] P. Maes. Situated agents can have goals. In P. Maes, editor, *Designing Autonomous Agents*, pages 49–70. The MIT Press: Cambridge, MA, 1990.

[39] P. Maes. Agents that reduce work and information overload. *Communications of the ACM*, 37(7):31–40, July 1994.

[40] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems — Safety*. Springer-Verlag: Heidelberg, Germany, 1995.

[41] Z. Manna and P. Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 6(1):68–93, January 1984.

[42] J. J. Ch. Meyer and R. J. Wieringa, editors. *Deontic Logic in Computer Science — Normative System Specification*. John Wiley & Sons, 1993.

[43] C. Morgan. *Programming from Specifications (second edition)*. Prentice Hall International: Hemel Hempstead, England, 1994.

[44] Inc. NeXT Computer. *Object-Oriented Programming and the Objective C Language*. Addison-Wesley: Reading, MA, 1993.

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

[46] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proceedings of the Sixteenth ACM Symposium on the Principles of Programming Languages (POPL)*, pages 179–190, January 1989.

[47] A. S. Rao and M. Georgeff. BDI Agents: from theory to practice. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95)*, pages 312–319, San Francisco, CA, June 1995.

[48] A. S. Rao and M. P. Georgeff. Modeling rational agents within a BDI-architecture. In R. Fikes and E. Sandewall, editors, *Proceedings of Knowledge Representation and Reasoning (KR&R-91)*, pages 473–484. Morgan Kaufmann Publishers: San Mateo, CA, April 1991.

[49] A. S. Rao and M. P. Georgeff. An abstract architecture for rational agents. In C. Rich, W. Swartout, and B. Nebel, editors, *Proceedings of Knowledge Representation and Reasoning (KR&R-92)*, pages 439–449, 1992.

[50] A. S. Rao and M. P. Georgeff. A model-theoretic approach to the verification of situated reasoning systems. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence (IJCAI-93)*, pages 318–324, Chambéry, France, 1993.

[51] A. S. Rao and M. P. Georgeff. Formal models and decision procedures for multi-agent systems. Technical Note 61, Australian AI Institute, Level 6, 171 La Trobe Street, Melbourne, Australia, June 1995.

[52] S. Rosenschein and L. P. Kaelbling. The synthesis of digital machines with provable epistemic properties. In J. Y. Halpern, editor, *Proceedings of the 1986 Conference on Theoretical Aspects of Reasoning About Knowledge*, pages 83–98. Morgan Kaufmann Publishers: San Mateo, CA, 1986.

[53] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, 1995.

[54] M. J. Schoppers. Universal plans for reactive robots in unpredictable environments. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence (IJCAI-87)*, pages 1039–1046, Milan, Italy, 1987.

[55] K. Segerberg. Bringing it about. *Journal of Philosophical Logic*, 18:327–347, 1989.

[56] Y. Shoham. *Reasoning About Change: Time and Causation from the Standpoint of Artificial Intelligence*. The MIT Press: Cambridge, MA, 1988.

[57] Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, 60(1):51–92, 1993.

[58] A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. *Journal of the ACM*, 32(3):733–749, 1985.

[59] R. M. Smullyan. *First-Order Logic*. Springer-Verlag: Heidelberg, Germany, 1968.

[60] M. Spivey. *The Z Notation (second edition)*. Prentice Hall International: Hemel Hempstead, England, 1992.

[61] M. Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, 1994.

[62] P. Wolper. The tableau method for temporal logic: An overview. *Logique et Analyse*, 110–111, 1985.

[63] M. Wooldridge. *The Logical Modelling of Computational Multi-Agent Systems*. PhD thesis, Department of Computation, UMIST, Manchester, UK, October 1992. (Also available as Technical Report MMU–DOC–94–01, Department of Computing, Manchester Metropolitan University, Chester St., Manchester, UK).

[64] M. Wooldridge. This is MYWORLD: The logic of an agent-oriented testbed for DAI. In M. Wooldridge and N. R. Jennings, editors, *Intelligent Agents: Theories, Architectures, and Languages (LNAI Volume 890)*, pages 160–178. Springer-Verlag: Heidelberg, Germany, January 1995.

[65] M. Wooldridge and M. Fisher. A decision procedure for a temporal belief logic. In D. M. Gabbay and H. J. Ohlbach, editors, *Temporal Logic — Proceedings of the First International Conference (LNAI Volume 827)*, pages 317–331. Springer-Verlag: Heidelberg, Germany, July 1994.

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