AGENTS AND SOFTWARE ENGINEERING

Michael Wooldridge
Queen Mary and Westfield College, University of London
London E1 4NS, United Kingdom
M.J.Wooldridge@qmw.ac.uk

Abstract

Software engineers continually strive to develop tools and techniques to manage the complexity that is inherent in software systems. In this article, we argue that *intelligent agents* and *agent-based systems* are just such tools. Following a discussion on the subject of what makes software complex, we introduce intelligent agents as software structures capable of making "rational decisions". Such rational decision-makers are well-suited to the construction of certain types of software, which mainstream software engineering has had little success with. We then go on to examine a number of prototype techniques proposed for engineering agent systems, including formal specification and verification methods for agent systems, and techniques for implementing agent specifications.

1 Introduction

It has long been known that software development is inherently difficult — that software has certain essential characteristics that make it hard to develop efficient, robust, correct software. Similarly, it has also been recognised that certain *types* of software system are harder to build than others. The discipline of software engineering is ultimately about understanding and mastering this inherent complexity, with the goal of making software development an everyday engineering task. Over its three decade history, software engineering has developed an increasingly powerful array of tools with which to tackle the complexity of software systems, of which the most recent additions are the notions of an *intelligent agent* and *multi-agent system* [24]. In brief, the aim of this article is to summarise why agents are perceived to be an important new development in software engineering, and then to review the various techniques and formalisms that have been developed for engineering agent-based systems. To this end, the article begins in the following section by attempting to identify some of the main characteristics of complex software systems. Section 3 then defines what we mean by the term "agent", and summarises why such agents might be appropriate for engineering complex software systems. In section 4, we describe agent-oriented specification techniques, focussing in particular on the requirements that an agent-oriented specification framework will have. In section 5, we discuss how such specifications can be implemented, either by directly executing them, or else by automatically synthesising executable systems from specifications. Section 6 discusses how implemented systems may be verified, to determine whether or not they satisfy their specifications. Finally, in section 7, we conclude with some comments on future issues for agent-oriented software engineering.

Note that sections 4 through to 6 include some material from [23], where a fuller examination of, in particular, the specification, implementation, and verification of agent-based systems may be found.

2 What Makes Complex Software?

As we noted in section 1, certain types of software system are harder to successfully engineer than others. Note we are not referring here to *algorithmic* complexity, in the sense of NP-completeness and Cook's theorem. Rather, we mean the complexity of system specification, design, and construction from a *software engineering* perspective. In order to do understand what makes certain types of software more complex than others, we will start from the abstract view of software systems presented in Figure 1. The idea is that any software system can be viewed as a function that is embedded within some environment, that takes input from the environment, and produces output that affects the environment. The environment is often software (e.g., an operating system such as UNIX or Windows 95), so that the actions the agent performs take the form of software operations such as writing to a file. However, if the system is embedded within a physical environment, then the actions the agent performs will correspond to "real world" actions, such as picking up and moving objects.

Given this abstract view, we can examine the factors that affect the complexity of engineering such software systems

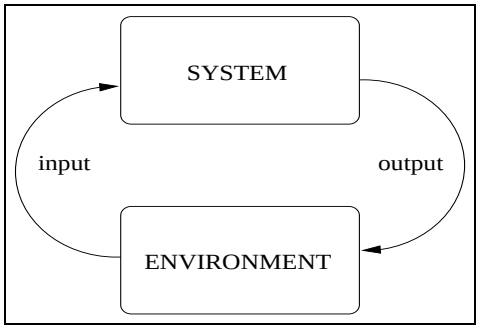

Figure 1: An abstract view of software systems.

along at least three dimensions:

- *The nature of the system's environment.*

  Whether or not the system's environment is *dynamic* or *static*, *accessible* or *inaccessible*, and *deterministic* or *non-deterministic* will affect the complexity of the software development process for systems situated in that environment [20, p46]. In general, environments that are dynamic, inaccessible, and non-deterministic will pose greater problems for the software developer than static, accessible, deterministic ones.

- *The nature of the interaction between the system and its environment.*

  The simplest general form of interaction with an environment is where a system takes some input from the environment, generates some output as a function of this input, and then terminates. Compilers are a classic example of such *functional* systems. Functional systems can be specified using pre- and post-condition formalisms, and many techniques (e.g., top-down stepwise refinement) are available to design and implement them. A more general, and typically more complex type of interaction is where a system is required to maintain an *ongoing*, *non-terminating* relationship with its environment. Examples include control systems and computer operating systems. It has long been recognised that the engineering of such *reactive* systems is hard (see, e.g., [13]). Note that reactive systems typically contain a number of reactive sub-systems, which interact with one-another in order to generate the global system behaviour.

- *The nature of the system's specification.*

  The simplest general sort of specification for a system is a predicate over programs: either a program satisfies the specification, or it doesn't. This is the model of specifications that is implicit within the formal specification community, as typified by work on the Z and VDM specification languages. The specification predicate induces an equivalence relation over the set of possible programs, whereby all those programs that satisfy the specification are considered equally good, and all those programs that fail to satisfy it are considered equally bad. A more complex general type of specification takes the form of *maximising payoff*: a system π is considered preferable to π' if the expected payoff by executing π is greater than the expected payoff by executing π'. Systems that are built to such specifications are essentially *rational decision makers*.

Traditional software engineering techniques have proved to be successful when directed to the construction of functional systems with simple predicate specifications, that are situated in static, accessible, deterministic environments. However, the construction of reactive systems that are situated in dynamic, inaccessible, non-deterministic environments with payoff-oriented specifications is an essentially open problem. This paper proceeds from the claim that in order to engineer such systems, we need new software development techniques and tools. This is not a very contentious claim: the fact that developing such systems is hard is well-known in mainstream software engineering. The question is, what techniques might be appropriate for this task? The second claim that this paper makes is that the *decision making* required of such systems closely resembles the *practical reasoning* that humans engage in every day. The discipline that studies the engineering of such computational practical reasoners is the field of *intelligent agents* [24]. The third claim that this paper makes is therefore that the technology of intelligent agents is a promising candidate for the engineering of such complex software systems.

3 Agent-Based Systems

By an *agent-based system*, we mean one in which the key abstraction used is that of an *agent*. By an *agent*, we mean a system that enjoys the following properties [24, pp116–118]:

- *autonomy*: agents encapsulate some state (which 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*, and typically have the ability to engage in social activities (such as cooperative problem solving or negotiation) in order to achieve their goals.

One of the main problems in developing an agent system is that of obtaining a *rational balance* between the tendency of the agent to react to environmental changes and its tendency to act towards its goals. It is easy to build agents that *only* react to their environment, and it is also easy to build systems that *only* act towards their goals. But building a system that gets an appropriate balance between these extremes is hard [24]. Agents that balance these two kinds of behaviour can be understood as *practical reasoning systems*, in much the sense that we discussed above. They are thus well suited to operate in the kinds of environment that traditional software engineering has not been successful at dealing with.

Now that we understand what an agent is, we can begin to look at *software engineering* for agent-based systems. Thus, in the following sections, we examine what specifications for agent systems might look like, how to implement such specifications, and finally, how to verify that implemented systems do in fact satisfy their specifications.

4 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 as practical reasoning systems that we 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 [5, 24]. 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 which explains how these aspects of agency interact to effect the mapping from sensor input to effector output (as shown in Figure 1) 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., [24] for extensive references). Two of the best known such logical frameworks are the Cohen-Levesque theory of intention [4], and the Rao-Georgeff belief-desire-intention model [16]. 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 agents internal state [24]. Comparatively few serious attempts have been made to specify real agent systems using such logics — see, e.g., [8] for one such attempt.

5 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:

1. somehow directly execute or animate the abstract specification; or
2. 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.

5.1 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 [7]. 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 [1].

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 [16]) are based on considerably more complex logics. In particular, they are usually based on a semantic framework known as *possible worlds* [2]. 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 interpretation 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, which although essentially a BDI system, has a simple computational semantics [15].

5.2 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 labeled graphs of some kind), and automata-like finite state machines. For example, Pnueli and Rosner [14] 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 [19]. They use an epistemic logic (i.e., a logic of *knowledge* [6]) to specify the perception component of intelligent agent systems. They then used an technique based on constructive proof to directly synthesise automata from these specifications [18].

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 offline 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.

6 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.

6.1 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 [10]. 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 [12]. The idea is that the computations of reactive systems are infinite sequences, which 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 [22], 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 be-

cause 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 [21], 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 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, 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).

6.2 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. 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; 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 ([3]) can be done in polynomial time, whereas the proof problem for most modal logics is quite complex.

In [17], 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 [9]. 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.

7 Conclusions

In this article, I have given a summary of why agents are perceived to be a significant technology for software engineering, and also of the main techniques for the specification, implementation, and verification of agent systems. 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 [11]), almost no attention has been given to the former (but see [25]).

References

[1] 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: Berlin, Germany, June 1989.

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

[3] 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: Berlin, Germany, 1981.

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

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

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

[7] M. Fisher. An alternative approach to concurrent theorem proving. In J. Geller, H. Kitano, and C. B. Suttner, editors, *Parallel Processing in Artificial Intelligence 3*, pages 209–230. Elsevier Science Publishers B.V.: Amsterdam, The Netherlands, 1997.

[8] M. Fisher and M. Wooldridge. On the formal specification and verification of multi-agent systems. *International Journal of Cooperative Information Systems*, 6(1):37–65, 1997.

[9] 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*, pages 151–176. Academic Press, 1991.

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

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

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

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

[14] 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.

[15] A. S. Rao. 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 Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World, (LNAI Volume 1038)*, pages 42–55. Springer-Verlag: Berlin, Germany, 1996.

[16] 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.

[17] 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.

[18] 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.

[19] S. J. Rosenschein and L. P. Kaelbling. A situated view of representation and control. In P. E. Agre and S. J. Rosenschein, editors, *Computational Theories of Interaction and Agency*, pages 515–540. The MIT Press: Cambridge, MA, 1996.

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

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

[22] M. Wooldridge. *The Logical Modelling of Computational Multi-Agent Systems*. PhD thesis, Department of Computation, UMIST, Manchester, UK, October 1992.

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

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

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