Agent-Oriented Software Engineering: The State of the Art

Michael Wooldridge[†] and **Paolo Ciancarini**[*]

[†] Department of Computer Science
University of Liverpool
Liverpool L69 7ZF, UK
M.J.Wooldridge@csc.liv.ac.uk

[*] Dipartimento di Scienze dell'Informazione
University of Bologna
Mura Anteo Zamboni 7, 47127 Bologna, Italy
ciancarini@cs.unibo.it

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 *multi-agent systems* are just such tools. We begin by reviewing what is meant by the term "agent", and contrast agents with objects. We then go on to examine a number of prototype techniques proposed for engineering agent systems, including methodologies for agent-oriented analysis and design, formal specification and verification methods for agent systems, and techniques for implementing agent specifications.

1 Introduction

Over the past three decades, software engineers have derived a progressively better understanding of the characteristics of complexity in software. It is now widely recognised that *interaction* is probably the most important single characteristic of complex software. Software architectures that contain many dynamically interacting components, each with their own thread of control, and engaging in complex coordination protocols, are typically orders of magnitude more complex to correctly and efficiently engineer than those that simply compute a function of some input through a single thread of control.

Unfortunately, it turns out that many (if not most) real-world applications have precisely these characteristics. As a consequence, a major research topic in computer science over at least the past two decades has been the development of tools and techniques to model, understand, and implement systems in which interaction is the norm.

Many researchers now believe that in future, computation itself will be understood as chiefly as a process of interaction. This has in turn led to the search for new computational abstractions, models, and tools with which to conceptualise and implement interacting systems.

Since the 1980s, software agents and multi-agent systems have grown into what is now one of the most active areas of research and development activity in computing generally. There are many reasons for the current intensity of interest, but certainly one of the most important is that the concept of an agent as an autonomous system, capable of interacting with other agents in order to satisfy its design objectives, is a natural one for software designers. Just as we can understand many systems as being composed of essentially passive objects, which have state, and upon which we can perform operations, so we can understand many others as being made up of interacting, semi-autonomous agents.

Our aim in this article is to survey the state of the art in agent-oriented software engineering. The article is structured as follows:

- in the sub-sections that follows, we provide brief introductions to agents and multi-agent systems, and comment on the relationship between agents and objects (in the sense of object-oriented programming);
- in section 2, we survey some preliminary *methodologies* for engineering multi-agent systems — these methodologies provide structured but non-mathematical approaches to the analysis and design of agent systems, and for the most part take inspiration either from object-oriented analysis and design methodologies or from knowledge-engineering approaches; and finally,
- in section 3, we comment on the use of *formal* methods for engineering multi-agent systems.

We conclude the main text of the article with a brief discussion of open problems, challenges, and issues that must be addressed if agents are to achieve their potential as a software engineering paradigm. In an appendix, we provide pointers to further information about agents.

1.1 What are Agent-Based Systems?

Before proceeding any further, it is important to gain an understanding of exactly what we mean by an agent-based system. 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 [50]), but arguably the greatest potential lies in the application of *multi*-agent systems [5]. By an *agent*, we mean a system that enjoys the following properties [75, pp.116–118]:

- *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* [28], and typically have the ability to engage in social activities (such as cooperative problem solving or negotiation) in order to achieve their goals.

These properties are more demanding than they might at first appear. To see why, let us consider them in turn. First, consider *pro-activeness*: goal directed behavior. It is not hard to build a system that exhibits goal directed behavior — we do it every time we write a procedure in Pascal, a function in C, or a method in Java. When we write such a procedure, we describe it in terms of the *assumptions* on which it relies (formally, its *pre-condition*) and the *effect* it has if the assumptions are valid (its *post-condition*). The effects of the procedure are its *goal*: what the author of the software intends the procedure to achieve. If the pre-condition holds when the procedure is invoked, then we expect that the procedure will execute *correctly*: that it will terminate, and that upon termination, the post-condition will be true, i.e., the goal will be achieved. This is goal directed behavior: the procedure is simply a plan or recipe for achieving the goal. This programming model is fine for many environments. For example, its works well when we consider *functional systems* — those that simply take some input x, and produce as output some some function $f(x)$ of this input. Compilers are a classic example of functional systems.

But for non-functional systems, this simple model of goal directed programming is not acceptable, as it makes an important limiting assumption. It assumes that the environment *does not change* while the procedure is executing. If the environment does change, and in particular, if the assumptions (pre-condition) underlying the procedure become false while the procedure is executing, then the behavior of the procedure may not be defined — often, it will simply crash. Similarly, it is assumed that the goal, that is, the reason for executing the procedure, remains valid at least until the procedure terminates. If the goal does *not* remain valid, then there is simply no reason to continue executing the procedure.

In many environments, neither of these assumptions are valid. In particular, in domains that are *too complex* for an agent to observe completely, that are *multi-agent* (i.e., they are populated with more than one agent that can change the environment), or where there is *uncertainty* in the environment, these assumptions are not reasonable. In such environments, blindly executing a procedure without regard to whether the assumptions underpinning the procedure are valid is a poor strategy. In such dynamic environments, an agent must be *reactive*, in just the way that we described above. That is, it must be responsive to events that occur in its environment, where these events affect either the agent's goals or the assumptions which underpin the procedures that the agent is executing in order to achieve its goals.

As we have seen, building purely goal directed systems is not hard. Similarly, building *purely reactive* systems — ones that *continually* respond to their environment — is also not difficult; we can implement them as lookup tables that simply match environmental stimuli to action responses. However, what turns out to be very hard is building a system that achieves an effective *balance* between goal-directed and reactive behavior. We want agents that will attempt to achieve their goals systematically, perhaps by making use of complex procedure-like recipes for action. But we don't want our agents to continue blindly executing these procedures in an attempt to achieve a goal either when it is clear that the procedure will not work, or when the goal is for some reason no longer valid. In such circumstances, we want our agent to be able to react to the new situation, in time for the reaction to be of some use. However, we do not want our agent to be *continually* reacting, and hence never focussing on a goal long enough to actually achieve it.

On reflection, it should come as little surprise that achieving a good balance between goal directed and reactive behavior is hard. After all, it is comparatively rare to find humans that do this very well. How many of us have had a manager who stayed blindly focussed on some project long after the relevance of the project was passed, or it was clear that the project plan was doomed to failure? Similarly, how many have encountered managers who seem unable to stay focussed at all, who flit from one project to another without ever managing to pursue a goal long enough to achieve *anything*? This problem — of effectively integrating goal-directed and reactive behavior — is one of the key problems facing the agent designer. As we shall see, a great many proposals have been made for how to build agents that can do this — but the problem is essentially still open.

Finally, let us say something about *social ability*, the final component of flexible autonomous action as defined here. In one sense, social ability is trivial: every day, millions of computers across the world routinely exchange information with both humans and other computers. But the ability to exchange bit streams is not really social ability. Consider that in the human world, comparatively few of our meaningful goals can be achieved without the *cooperation* of other people, who cannot be assumed to *share* our goals — in other words, they are themselves autonomous, with their own agenda to pursue. This type of social ability — involving the ability to dynamically negotiate and coordinate — is much more complex, and much less well understood, than simply the ability to exchange bitstreams.

An obvious question to ask is why agents and multi-agent systems are seen as an important new direction in software engineering. There are several reasons [40, pp.6–10]:

– *Natural metaphor*.
  Just as the many domains can be conceived of consisting of a number of interacting but essentially passive *objects*, so many others can be conceived as interacting, active, purposeful *agents*. For example, a scenario currently driving much R&D activity in the agents field is that of software agents that buy and sell goods via the Internet on behalf of some users. It is natural to view the software participants in such transactions as (semi-)autonomous agents.
– *Distribution of data or control*.
  For many software systems, it is not possible to identify a single locus of control: instead, overall control of the systems is distributed across a number computing nodes, which are frequently geographically distributed. In order to make such systems work effectively, these nodes must be capable of autonomously interacting with each other — they must agents.
– *Legacy systems*.

A natural way of incorporating legacy systems into modern distributed information systems is to *agentify* them: to "wrap" them with an agent layer, that will enable them to interact with other agents.
- *Open systems*.
  Many systems are *open* in the sense that it is impossible to know at design time exactly what components the system will be comprised of, and how these components will be used to interact with one-another. To operate effectively in such systems, the ability to engage in flexible autonomous decision-making is critical.

1.2 On the Relationship between Agents and Objects

Programmers familiar with object-oriented approaches often fail to see anything novel or new in the idea of agents. When one stops to consider the relative properties of agents and objects, this is perhaps not surprising. Objects are defined as computational entities that *encapsulate* some state, are able to perform actions, or *methods* on this state, and communicate by message passing. There are clearly close links between agents and objects, which are made stronger by our tendency to anthropomorphisize objects. For example, 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. [37, p.7]

While there are obvious similarities, there are also significant differences between agents and objects. The first is in the degree to which agents and objects are autonomous. Recall that the defining characteristic of object-oriented programming is the principle of encapsulation — the idea that objects can have control over their own internal state. In programming languages like Java, we can declare instance variables (and methods) to be `private`, meaning they are only accessible from within the object. (We can of course also declare them `public`, meaning that they can be accessed from anywhere, and indeed we must do this for methods so that they can be used by other objects. But the use of `public` instance variables is usually considered poor programming style.) In this way, an object can be thought of as exhibiting autonomy over its state: it has control over it. But an object does not exhibit control over it's *behavior*. That is, if an object has a public method `m`, then other objects can invoke `m` whenever they wish — once an object has made a method `public`, then it subsequently has no control over whether or not that method is executed.

Of course, an object *must* make methods available to other objects, or else we would be unable to build a system out of them. This is not normally an issue, because if we build a system, then we design the objects that go in it, and they can thus be assumed to share a "common goal". But in many types of multi-agent system, (in particular, those that contain agents built by different organisations or individuals), no such common goal can be assumed. It cannot be for granted that an agent i will execute an action (method) a just because another agent j wants it to — a may not be in the best interests of i. We thus do not think of agents as invoking methods upon one-another, but rather as *requesting* actions to be performed. If j requests i to perform a, then i may perform the action or it may not. The locus of control with respect to the decision about whether to execute an action is thus different in agent and object systems. In the object-oriented case, the decision lies with the object that invokes the method. In the agent case, the decision lies with the agent that receives the request. This distinction between objects and agents has been nicely summarized in the following slogan: *Objects do it for free; agents do it because they want to*.

The second important distinction between object and agent systems is with respect to the notion of flexible (reactive, pro-active, social) autonomous behavior. The standard object model has nothing whatsoever to say about how to build systems that integrate these types of behavior. One could point out that we can build object-oriented programs that *do* integrate these types of behavior. And indeed we can, but this argument misses the point, which is that the standard object-oriented programming model has nothing to do with these types of behavior.

The third important distinction between the standard object model and our view of agent systems is that agents are each considered to have their own thread of control. Agents are assumed to be continually active, and typically are engaged in an infinite loop of observing their environment, updating their internal state, and selecting and executing an action to perform. In contrast, objects are assumed to be quiescent for most of the time, becoming active only when another object requires their services by dint of method invocation.

Of course, a lot of work has recently been devoted to *concurrency* in object-oriented programming. For example, the Java language provides built-in constructs for multi-threaded programming. There are also many programming languages available (most of them admittedly prototypes) that were specifically designed to allow concurrent object-based programming. But such languages do not capture the idea we have of agents as *autonomous* entities. Perhaps the closest that the object-oriented community comes is in the idea of *active objects*:

> An active object is one that encompasses its own thread of control [...]. Active objects are generally autonomous, meaning that they can exhibit some behavior without being operated upon by another object. Passive objects, on the other hand, can only undergo a state change when explicitly acted upon. [6, p.91]

Thus active objects are essentially agents that do not necessarily have the ability to exhibit flexible autonomous behavior.

To summarize, the traditional view of an object and our view of an agent have at least three distinctions:

- agents embody stronger notion of autonomy than objects, and in particular, they decide for themselves whether or not to perform an action on request from another agent;
- agents are capable of flexible (reactive, pro-active, social) behavior, and the standard object model has nothing to say about such types of behavior;
- a multi-agent system is inherently multi-threaded, in that each agent is assumed to have at least one thread of control.

2 Agent-Oriented Analysis and Design

The first main strand of work we consider on approaches to developing agent systems involves principled but *informal* development methodologies for the analysis and design of agent-based system. These can be broadly divided into two groups:

- those that take their inspiration from object-oriented development, and either extend existing OO methodologies or adapt OO methodologies to the purposes of AOSE [10, 45, 77, 54, 18, 3, 44, 56, 70];
- those that adapt knowledge engineering or other techniques [8, 49, 36, 16].

In the remainder of this section, we review some representative samples of this work. As representatives of the first category, we survey the AAII methodology of Kinny et al [45], the Gaia methodology of Wooldridge et al [77], and summarise work on adapting UML [54, 18, 3]. As representatives of the second category, we survey the Cassiopeia methodology of Collinot et al [16], the DESIRE framework of Treur et al [8], and the use of Z for specifying agent systems [49].

Kinny et al: The AAII Methodology The Australian AI Institute (AAII) has been developing agent-based systems for a decade. The primary development environment in which this work has been carried out is the belief-desire-intention technology [74] of the Procedural Reasoning System (PRS) and its successor, the Distributed Multi-Agent Reasoning System (DMARS) [62]. The PRS, originally developed at Stanford Research Institute, was the first agent architecture to explicitly embody the belief-desire-intention paradigm, and has proved to be the most durable agent architecture developed to date. It has been applied in several of the most significant multi-agent applications so far built, including an air-traffic control system called OASIS that is currently undergoing field trials at Sydney airport, a simulation system for the Royal Australian Air Force called SWARMM, and a business process management system called SPOC (Single Point of Contact), that is currently being marketed by Agentis Solutions [29]. The AAII methodology for agent-oriented analysis and design was developed as a result of experience gained with these major applications. It draws primarily upon object-oriented methodologies, and enhances them with some agent-based concepts. The methodology itself is aimed at the construction of a set of models which, when fully elaborated, define an agent system specification.

The AAII methodology provides both *internal* and *external* models. The external model presents a system-level view: the main components visible in this model are agents themselves. The external model is thus primarily concerned with agents and the relationships between them. It is not concerned with the internals of agents: how they are constructed or what they do. In contrast, the internal model is entirely concerned with the internals of agents: their beliefs, desires, and intentions.

The external model is intended to define inheritance relationships between agent classes, and to identify the instances of these classes that will appear at run-time. It is itself composed of two further models: the *agent model* and the *interaction model*. The agent model is then further divided into an *agent class model* and an *agent instance model*. These two models define the agents and agent classes that can appear, and relate these classes to one-another via inheritance, aggregation, and instantiation relations. Each agent class is assumed to have at least three attributes, for beliefs, desires, and intentions. The analyst is able to define how these attributes are overridden during inheritance. For example, it is assumed that by default, inherited intentions have less priority than those in sub-classes. The analyst may tailor these properties as desired.

Details of the internal model are not given, but it seems clear that developing an internal model corresponds fairly closely to implementing a PRS agent, i.e., designing the agent's belief, desire, and intention structures.

The AAII methodology is aimed at elaborating the models described above. It may be summarised 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.
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.

Wooldridge et al: Gaia The Gaia[1] methodology is intended to allow an analyst to go systematically from a statement of requirements to a design that is sufficiently detailed that it can be implemented directly. Note that we view the requirements capture phase as being independent of the paradigm used for analysis and design. In applying Gaia, the analyst moves from abstract to increasingly concrete concepts. Each successive move introduces greater implementation bias, and shrinks the space of possible systems that could be implemented to satisfy the original requirements statement. (See [42, pp.216-222] for a discussion of implementation bias.) Analysis and design can be thought of as a process of developing increasingly detailed *models* of the system to be constructed.

[1] The name comes from the Gaia hypothesis put forward by James Lovelock, to the effect that all the organisms in the earth's biosphere can be viewed as acting together to regulate the earth's environment.

Abstract Concepts	Concrete Concepts
Roles	Agent Types
Permissions	Services
Responsibilities	Acquaintances
Protocols	
Activities	
Liveness properties	
Safety properties	

Table 1. Abstract and concrete concepts in Gaia

Gaia borrows some terminology and notation from object-oriented analysis and design, (specifically, FUSION [15]). However, it is not simply a naive attempt to apply such methods to agent-oriented development. Rather, it provides an agent-specific set of concepts through which a software engineer can understand and model a complex system. In particular, Gaia encourages a developer to think of building agent-based systems as a process of *organisational design*.

The main Gaian concepts can be divided into two categories: *abstract* and *concrete*; abstract and concrete concepts are summarised in Table 1. Abstract entities are those used during analysis to conceptualise the system, but which do not necessarily have any *direct* realisation within the system. Concrete entities, in contrast, are used within the design process, and will typically have direct counterparts in the run-time system.

The objective of the analysis stage is to develop an understanding of the system and its structure (without reference to any implementation detail). In the Gaia case, this understanding is captured in the system's *organisation*. An organisation is viewed as a collection of roles, that stand in certain relationships to one another, and that take part in systematic, institutionalised patterns of interactions with other roles.

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

A role is defined by four attributes: *responsibilities*, *permissions*, *activities*, and *protocols*. *Responsibilities* determine functionality and, as such, are perhaps the key attribute associated with a role. An example responsibility associated with the role of company president might be calling the shareholders meeting every year. Responsibilities are divided into two types: *liveness properties* and *safety properties* [57]. Liveness properties intuitively state that "something good happens". They describe those states of affairs that an agent must bring about, given certain environmental conditions. In contrast, safety properties are *invariants*. Intuitively, a safety property states that "nothing bad happens" (i.e., that an acceptable state of affairs is maintained across all states of execution). An example might be "ensure the reactor temperature always remains in the range 0-100".

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

The *activities* of a role are computations associated with the role that may be carried out by the agent without interacting with other agents. Activities are thus "private" actions, in the sense of [65].

Finally, a role is also identified with a number of *protocols*, which define the way that it can interact with other roles. For example, a "seller" role might have the protocols "Dutch auction" and "English auction" associated with it; the Contract Net Protocol is associated with the roles "manager" and "contractor" [66].

Odell et al: Agent UML Over the past two decades, many different notations and associated methodologies have been developed within the object-oriented development community (see, e.g., [6, 64, 15]). Despite many similarities between these notations and methods, there were nevertheless many fundamental inconsistencies and differences. The Unified Modelling Language — UML — is an attempt by three of the main figures behind object-oriented analysis and design (Grady Booch, James Rumbaugh, and Ivar Jacobson) to develop a single notation for modelling object-oriented systems [7]. It is important to note that UML is *not* a methodology; it is, as its name suggests, a language for documenting models of systems; associated with UML is a methodology known as the Rational Unified Process [7, pp.449–456].

The fact that UML is a de facto standard for object-oriented modelling promoted its rapid takeup. When looking for agent-oriented modelling languages and tools, many researchers felt that UML was the obvious place to start [54, 18, 3]. The result has been a number of attempts to adapt the UML notation for modelling agent systems. Odell and colleagues have discussed several ways in which the UML notation might usefully be extended to enable the modelling of agent systems [54, 3]. The proposed modifications include:

– support for expressing concurrent threads of interaction (e.g., broadcast messages), thus enabling UML to model such well-known agent protocols as the Contract Net [66];
– a notion of "role" that extends that provided in UML, and in particular, allows the modelling of an agent playing many roles.

Both the Object Management Group (OMG) [55], and the Foundation for Intelligent Physical Agents (FIPA) [27] are currently supporting the development of UML-based notations for modelling agent systems, and there is therefore likely to be considerable work in this area.

Treur et al: DESIRE In an extensive series of papers (see, e.g., [8, 19]), Treur and colleagues have described the DESIRE framework. DESIRE is a framework for the design and formal specification of compositional systems. As well as providing a graphical notation for specifying such compositional systems, DESIRE has associated with it a graphical editor and other tools to support the development of agent systems.

Collinot et al: Cassiopeia In contrast to Gaia and the AAII methodology, the Cassiopeia method proposed by Collinot et al is essentially *bottom up* in nature [16]. Essentially, with the Cassiopeia method, one starts from the *behaviours* required to carry out some task; this is rather similar to the behavioural view of agents put forward by Brooks and colleagues [9]. Essentially, the methodology proposes three steps:

1. identify the *elementary behaviours* that are implied by the overall system task;
2. identify the *relationships* between elementary behaviours;
3. identify the *organisational behaviours* of the system, for example, the way in which agents form themselves into groups.

Collinot et al illustrate the methodology by way of the design of a RoboCup soccer team (see [38]).

Luck and d'Inverno: Agents in Z Luck and d'Inverno have developed an agent specification framework in the Z language [68], although the types of agents considered in this framework are somewhat different from those discussed above [48, 49]. 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 [49], 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. [49]

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[2].

2.1 Discussion

The predominant approach to developing methodologies for multi-agent systems is to adapt those developed for object-oriented analysis and design: hence the AAII methodology takes inspiration from Rumbaugh's work, Gaia takes inspiration from FUSION, and so on. There are obvious advantages to such an approach, the most obvious being that the concepts, notations, and methods associated with object-oriented analysis and design (and UML in particular) are increasingly familiar to a mass audience of software engineers. However, there are several disadvantages. First, the kinds of *decomposition* that object-oriented methods encourage is at odds with the kind of decomposition that *agent oriented* design encourages. Put crudely, agents are more coarse-grained computational objects than are agents; they are typically assumed to have the computational resources of a UNIX process, or at least a Java thread. Agent systems implemented using object-oriented programming languages will typically contain many objects (perhaps millions), but will contain far fewer agents. A good agent oriented design methodology would encourage developers to achieve the correct decomposition of entities into either agents or objects.

Note that an alternative would be to model every entity in a system as an agent. However, while this may be in some sense conceptually clean, does not lead to efficient systems (see the discussion in [76]). The situation reflects the treatment of integer data types in object-oriented programming languages; in "pure" OO languages, all data types, including integers, are objects. However, viewing such primitive data types as objects, while ensuring a consistent treatment of data, is not terribly efficient, and for this reason, more pragmatic OO languages (such as Java) do not treat integers, booleans, and the like as objects.

Another problem is that object-oriented methodologies simply do not allow us to capture many aspects of agent systems; for example, it is hard to capture in object models such notions as an agent pro-actively generating actions or dynamically reacting to changes in their environment, still less how to effectively cooperate and negotiate with other self-interested agents. The extensions to UML proposed by Odell et al [54, 18, 3] address some, but by no means all of these deficiencies. At the heart of the problem is the problem of the relationship between agents and objects, which has not yet been satisfactorily resolved.

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

Note that a valuable survey of methodologies for agent-oriented software engineering can be found in [35].

3 Formal Methods for AOSE

One of the most active areas of work in agent-oriented software engineering has been on the use of *formal methods* (see, e.g., [75] for a survey). Broadly speaking, formal methods play three roles in software engineering:

- in the *specification* of systems;
- for *directly programming* systems; and
- in the *verification* of systems.

In the subsections that follow, we consider each of these roles in turn. Note that these subsections pre-suppose some familiarity with formal methods, and logic in particular.

3.1 Formal Methods in 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 [17, 75, 74]. 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 refer to a theory which explains how these aspects of agency interact to generate the behaviour of an agent as 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., [75] for extensive references). Two of the best known such logical frameworks are the Cohen-Levesque theory of intention [14], and the Rao-Georgeff belief-desire-intention model [60, 74]. 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 [75]. Comparatively few serious attempts have been made to specify real agent systems using such logics — see, e.g., [26] for one such attempt.

A specification expressed in such a logic would be a formula φ. The idea is that such a specification would express the desirable behavior of a system. To see how this might work, consider the following, intended to form part of a specification of a process control system.

if
  i believes valve 32 is open
then
  i should intend that j should believe valve 32 is open

Expressed in the BDI logic developed in [74], this statement becomes the formula:

$$(\mathsf{Bel}\ i\ Open(valve32)) \Rightarrow (\mathsf{Int}\ i\ (\mathsf{Bel}\ j\ Open(valve32)))$$

It should be intuitively clear how a system specification might be constructed using such formulae, to define the intended behavior of a system.

One of the main desirable features of a software specification language is that it should not dictate *how* a specification will be satisfied by an implementation. The specification above has exactly this property: it does not dictate how agent i should go about making j aware that valve 32 is open. We simply expect i to behave as a rational agent given such an intention [74].

There are a number of problems with the use of languages such as for specification. The most worrying of these is with respect to their semantics. The semantics for the modal connectives (for beliefs, desires, and intentions) are given in the normal modal logic tradition of possible worlds [11]. So, for example, an agent's beliefs in some state are characterized 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 agent's desires in some state are characterized by a set of states that are consistent with the agent's 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 of 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.

Second, 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 agent's state, and any concrete computational model. As we shall see in later sections, this makes it 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 such as are to be taken seriously as *specification* languages, then this is a significant problem.

3.2 Formal Methods in Implementation

Specification is not (usually!) 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. directly execute or animate the abstract specification; or
3. translate or compile the specification into a concrete computational form using an automatic translation technique.

In the subsections that follow, we shall investigate each of these possibilities in turn.

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 subsystem specifications, which together satisfy the original specification. If these subsystems are still too abstract to be implemented directly, then they are also refined. The process recurses until the derived subsystems 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 [68] or VDM [42]. More usually, justification is by informal argument. Object-oriented analysis and design techniques, which also tend to be structured but informal, are also increasingly playing a role in the development of systems (see, e.g., [6]).

For *functional* systems, which simply compute a function of some input and then terminate, the refinement process is well understood, and comparatively straightforward. Such systems can be specified in terms of pre- and post-conditions (e.g., using Hoare logic [32]). 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 [53]. Part of the 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 agent systems, which fall into the category of Pnuelian reactive systems (see the discussion in chapter 1), refinement is not so straightforward. This is because such systems must be specified in terms of their *ongoing* behavior — they cannot be specified simply in terms of pre- and post-conditions. In contrast to pre- and post-condition formalisms, it is not so easy to determine what program structures are required to realize such specifications. As a consequence, researchers have only just begun to investigate refinement and design technique for agent-based systems.

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, which 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 logic-based AI hate. One should 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 $M \models \varphi$ [24]. If this could be 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 from doing so. The game is played by agent and environment taking 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.

Concurrent METATEM is a programming language for multiagent systems, that is based on the idea of directly executing linear time temporal logic agent specifications [25, 23]. 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 behavior 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 behavior 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).

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

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 [67].

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 [58] 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 characterizes 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 the universal path quantifier. 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., $O(2^{2^{c \cdot n}})$, 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.

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 program skeleton (based upon Hoare's CSP formalism [33]) that realizes the specification [52]. 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.

Perhaps the best-known example of this approach to agent development is the *situated automata* paradigm of Rosenschein and Kaelbling [63]. 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.

3.3 Formal 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.

Axiomatic Approaches: Deductive Verification. Axiomatic approaches to program verification were the first to enter the mainstream of computer science, with the work of Hoare in the late 1960s [32]. Axiomatic verification requires that we can take our concrete program, and from this program systematically derive a logical theory that represents the behavior 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 [51]. 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 multiagent environments. I shall review just one approach.

In [71], an axiomatic approach to the verification of multiagent systems was proposed. Essentially, the idea was to use a temporal belief logic to axiomatize the properties of two multiagent 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 behavior of a multiagent 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 behavior. A belief component in the logic allows us to capture the symbolic representations present within each agent.

The two multiagent programming languages that were axiomatized in the temporal belief logic were Shoham's AGENT0 [65], 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 modeled at stage (1).
3. The temporal belief logic was used to axiomatize the properties of a multiagent programming language. This axiomatization was then used to develop the program theory of a multiagent system.
4. The proof theory of the temporal belief logic was used to verify properties of the system (cf. [20]).

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).

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* [13]. 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 $M \models_L \varphi$. Verification by model checking has been studied in connection with temporal logic [13]. 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 [12] can be done in 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.

In [61], Rao and Georgeff present an algorithm for model checking BDI systems. More precisely, they give an algorithm for taking a logical model for their (propositional) BDI logic, 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 [13]. 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. Similar algorithms have been reported for BDI-like logics in [4].

The main problem with model-checking approaches for BDI is that it is not clear how the first step might be realized for BDI logics. Where does the logical model characterizing an agent actually come 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.

3.4 Discussion

This section is an updated and modified version of [73], which examined the possibility of using logic to engineer agent-based systems. Since this article was published, several other authors have proposed the use of agents in software engineering (see, e.g., [39]).

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 [49]. 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.

With respect to the possibility of directly executing agent specifications, a number of problems 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 (among 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 [1].

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 languages like , where we have not only a temporal dimension to the logic, but also modalities for referring to beliefs, desires, and so on. In general, models for these logics 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 [20, p.72]; richer logics, or course have more complex decision problems.

Despite these problems, the undoubted attractions of direct execution have led to a number of attempts to devise executable logic-based agent languages. Rao proposed an executable subset of BDI logic in his AGENTSPEAK(L) language [59]. Building on this work, Hindriks and colleagues developed the 3APL agent programming language [30, 31]. Lespérance, Reiter, Levesque, and colleagues developed the GOLOG language throughout the latter half of the 1990s as an executable subset of the situation calculus [46, 47]. Fagin and colleagues have proposed *knowledge-based programs* as a paradigm for executing logical formulae which contain epistemic modalities [20, 21]. Although considerable work has been carried out on the properties of knowledge-based programs, comparatively little research to date has addressed the problem of how such programs might be actually executed.

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 1,000, a synthesis algorithm that runs in exponential time when used off-line is no more useful than an execution algorithm that runs in exponential time on-line.

Another problem with respect to synthesis techniques is that they typically result in finite-state, automata-like machines, which are less powerful than Turing machines. In particular, the systems generated by the processes outlined above cannot modify their behavior 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 [50], learning is pretty much the *raison d'etre*. Attempts to address this issue are described in [43].

Turning to verification, axiomatic approaches 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 [72], is to model agent execution cycles as intervals over the real numbers, in the style of the temporal logic of reals [2]. The second problem is the difficulty of the proof problem for agent specification languages. The theoretical complexity of proof for many of these logics is quite daunting.

Hindriks and colleagues have used Plotkin's structured operational semantics to axiomatize their 3APL language [30, 31].

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.

4 Conclusions

Agent-oriented software engineering is at an early stage of evolution. While there are many good paper arguments to support the view that agents represent an important direction for software engineering, there is as yet a dearth of actual experience to underpin these arguments. Preliminary methodologies and software tools to support the deployment of agent systems are beginning to appear, but slowly. In this final section, we point to some of what we believe are the key obstacles that must be overcome in order for AOSE to become "mainstream":

- *Sorting out the relationship of agents to other software paradigms — objects in particular.*
  It is not yet clear how the development of agent systems will coexist with other software paradigms, such as object-oriented development.
- *Agent-oriented methodologies.*
  Although, as we have seen in this article, a number of preliminary agent-oriented analysis and design methodologies have been proposed, there is comparatively little consensus between these. In most cases, there is not even agreement on the kinds of concepts the methodology should support. The waters are muddied by the presence of UML as the predominant modelling language for object-oriented systems [7]: we suggested earlier that the kinds of concepts and notations supported by UML are not necessarily those best-suited to the development of agent systems.
- *Engineering for open systems.*
  We argued that agents are suitable for *open* systems. In such systems, we believe it is essential to be capable of reacting to unforeseen events, exploiting opportunities where these arise, and dynamically reaching agreements with system components whose presence could not be predicted at design time. However, it is difficult to know how to *specify* such systems; still less how to implement them. In short, we need a better understanding of how to engineer open systems.
- *Engineering for scalability.*
  Finally, we need a better understanding of how to safely and predictably engineer systems comprised of massive numbers of agents dynamically interacting with one-another in order to achieve their goals. Such systems seem prone to problems such as unstable/chaotic behaviours, feedback, and so on, and may fall prey to malicious behaviour such as viruses.

Appendix: How to Find Out More About Agents

There are now many introductions to intelligent agents and multiagent systems. Ferber [22] is an undergraduate textbook, although as its name suggests, this volume focussed on multiagent aspects rather than on the theory and practice of individual agents. A first-rate collection of articles introducing agent and multiagent systems is Weiß [69]. Two collections of research articles provide a comprehensive introduction to the field of autonomous rational agents and multiagent systems: Bond and Gasser's 1988 collection, *Readings in Distributed Artificial Intelligence*, introduces almost all the basic problems in the multiagent systems field, and although some of the papers it contains are now rather dated, it remains essential reading [5]; Huhns and Singh's more recent collection sets itself the ambitious goal of providing a survey of the whole of the agent field, and succeeds in this respect very well [34]. For a general introduction to the theory and practice of intelligent agents, see Wooldridge and Jennings [75], which focuses primarily on the theory of agents, but also contains an extensive review of agent architectures and programming languages. For a collection of articles on the applications of agent technology, see [41]. A comprehensive roadmap of agent technology was published as [40].

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. 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.
3. Bernhard Bauer, Jörg P. Müller, and James Odell. Agent UML: A formalism for specifying multiagent software systems. In P. Ciancarini and M. Wooldridge, editors, *Agent-Oriented Software Engineering — Proceedings of the First International Workshop (AOSE-2000)*. Springer-Verlag: Berlin, Germany, 2000.
4. M. Benerecetti, F. Giunchiglia, and L. Serafini. A model checking algorithm for multiagent systems. In J. P. Müller, M. P. Singh, and A. S. Rao, editors, *Intelligent Agents V (LNAI Volume 1555)*. Springer-Verlag: Berlin, Germany, 1999.
5. A. H. Bond and L. Gasser, editors. *Readings in Distributed Artificial Intelligence*. Morgan Kaufmann Publishers: San Mateo, CA, 1988.
6. G. Booch. *Object-Oriented Analysis and Design (second edition)*. Addison-Wesley: Reading, MA, 1994.
7. G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley: Reading, MA, 1999.
8. F. Brazier, B. Dunin-Keplicz, N. R. Jennings, and J. Treur. Formal specification of multi-agent systems: a real-world case. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95)*, pages 25–32, San Francisco, CA, June 1995.
9. R. A. Brooks. *Cambrian Intelligence*. The MIT Press: Cambridge, MA, 1999.
10. Birgit Burmeister. Models and methodologies for agent-oriented analysis and design. In Klaus Fischer, editor, *Working Notes of the KI'96 Workshop on Agent-Oriented Programming and Distributed Systems*. 1996. DFKI Document D-96-06.
11. B. Chellas. *Modal Logic: An Introduction*. Cambridge University Press: Cambridge, England, 1980.
12. 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.
13. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press: Cambridge, MA, 2000.
14. P. R. Cohen and H. J. Levesque. Intention is choice with commitment. *Artificial Intelligence*, 42:213–261, 1990.
15. D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes. *Object-Oriented Development: The* FUSION *Method*. Prentice Hall International: Hemel Hempstead, England, 1994.
16. Anne Collinot, Alexis Drogoul, and Philippe Benhamou. Agent oriented design of a soccer robot team. In *Proceedings of the Second International Conference on Multi-Agent Systems (ICMAS-96)*, pages 41–47, Kyoto, Japan, 1996.

17. D. C. Dennett. *The Intentional Stance*. The MIT Press: Cambridge, MA, 1987.
18. Ralph Depke, Reiko Heckel, and Jochen Malte Kuester. Requirement specification and design of agent-based systems with graph transformation, roles, and uml. In P. Ciancarini and M. Wooldridge, editors, *Agent-Oriented Software Engineering — Proceedings of the First International Workshop (AOSE-2000)*. Springer-Verlag: Berlin, Germany, 2000.
19. B. Dunin-Keplicz and J. Treur. Compositional formal specification of multi-agent systems. In M. Wooldridge and N. R. Jennings, editors, *Intelligent Agents: Theories, Architectures, and Languages (LNAI Volume 890)*, pages 102–117. Springer-Verlag: Berlin, Germany, January 1995.
20. R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. *Reasoning About Knowledge*. The MIT Press: Cambridge, MA, 1995.
21. R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. Knowledge-based programs. *Distributed Computing*, 10(4):199–225, 1997.
22. J. Ferber. *Multi-Agent Systems*. Addison-Wesley: Reading, MA, 1999.
23. 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: Berlin, Germany, July 1994.
24. M. Fisher. An introduction to executable temporal logic. *The Knowledge Engineering Review*, 11(1):43–56, 1996.
25. 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.
26. 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.
27. The Foundation for Intelligent Physical Agents. See `http://www.fipa.org/`.
28. M. R. Genesereth and S. P. Ketchpel. Software agents. *Communications of the ACM*, 37(7):48–53, July 1994.
29. M. P. Georgeff and A. S. Rao. A profile of the Australian AI Institute. *IEEE Expert*, 11(6):89–92, December 1996.
30. K. V. Hindriks, F. S. de Boer, W. van der Hoek, and J.-J. Ch. Meyer. Formal semantics for an abstract agent programming language. In M. P. Singh, A. Rao, and M. J. Wooldridge, editors, *Intelligent Agents IV (LNAI Volume 1365)*, pages 215–230. Springer-Verlag: Berlin, Germany, 1998.
31. K. V. Hindriks, F. S. de Boer, W. van der Hoek, and J.-J. Ch. Meyer. Control structures of rule-based agent languages. In J. P. Müller, M. P. Singh, and A. S. Rao, editors, *Intelligent Agents V (LNAI Volume 1555)*. Springer-Verlag: Berlin, Germany, 1999.
32. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, 1969.
33. C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21:666–677, 1978.
34. M. Huhns and M. P. Singh, editors. *Readings in Agents*. Morgan Kaufmann Publishers: San Mateo, CA, 1998.
35. C. A. Iglesias, M. Garijo, and J. C. Gonzalez. A survey of agent-oriented methodologies. In J. P. Müller, M. P. Singh, and A. S. Rao, editors, *Intelligent Agents V (LNAI Volume 1555)*. Springer-Verlag: Berlin, Germany, 1999.
36. Carlos Iglesias, Mercedes Garijo, José C. González, and Juan R. Velasco. Analysis and design of multiagent systems using MAS-CommonKADS. In M. P. Singh, A. Rao, and M. J. Wooldridge, editors, *Intelligent Agents IV (LNAI Volume 1365)*, pages 313–326. Springer-Verlag: Berlin, Germany, 1998.

37. NeXT Computer Inc. *Object-Oriented Programming and the Objective C Language*. Addison-Wesley: Reading, MA, 1993.
38. The Robot World Cup Initiative. See `http://www.RoboCup.org/`.
39. N. R. Jennings. Agent-based computing: Promise and perils. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI-99)*, pages 1429–1436, Stockholm, Sweden, 1999.
40. N. R. Jennings, K. Sycara, and M. Wooldridge. A roadmap of agent research and development. *Autonomous Agents and Multi-Agent Systems*, 1(1):7–38, 1998.
41. N. R. Jennings and M. Wooldridge, editors. *Agent Technology: Foundations, Applications and Markets*. Springer-Verlag: Berlin, Germany, 1998.
42. C. B. Jones. *Systematic Software Development using VDM (second edition)*. Prentice Hall, 1990.
43. L. P. Kaelbling. *Learning in Embedded Systems*. The MIT Press: Cambridge, MA, 1993.
44. Elizabeth A. Kendall. Agent software engineering with role modelling. In P. Ciancarini and M. Wooldridge, editors, *Agent-Oriented Software Engineering — Proceedings of the First International Workshop (AOSE-2000)*. Springer-Verlag: Berlin, Germany, 2000.
45. D. Kinny, M. Georgeff, and A. Rao. A methodology and modelling technique for systems of BDI agents. In W. Van de Velde and J. W. Perram, editors, *Agents Breaking Away: Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World, (LNAI Volume 1038)*, pages 56–71. Springer-Verlag: Berlin, Germany, 1996.
46. Y. Lésperance, H. J. Levesque, F. Lin, D. Marcu, R. Reiter, and R. B. Scherl. Foundations of a logical approach to agent programming. In M. Wooldridge, J. P. Müller, and M. Tambe, editors, *Intelligent Agents II (LNAI Volume 1037)*, pages 331–346. Springer-Verlag: Berlin, Germany, 1996.
47. H. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. Scherl. Golog: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31:59–84, 1996.
48. 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.
49. M. Luck, N. Griffiths, and M. d'Inverno. From agent theory to agent construction: A case study. In J. P. Müller, M. Wooldridge, and N. R. Jennings, editors, *Intelligent Agents III (LNAI Volume 1193)*, pages 49–64. Springer-Verlag: Berlin, Germany, 1997.
50. P. Maes. Agents that reduce work and information overload. *Communications of the ACM*, 37(7):31–40, July 1994.
51. Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems — Safety*. Springer-Verlag: Berlin, Germany, 1995.
52. 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.
53. C. Morgan. *Programming from Specifications (second edition)*. Prentice Hall International: Hemel Hempstead, England, 1994.
54. James Odell, H. Van Dyke Parunak, and Bernhard Bauer. Representing agent interaction protocols in UML. In P. Ciancarini and M. Wooldridge, editors, *Agent-Oriented Software Engineering — Proceedings of the First International Workshop (AOSE-2000)*. Springer-Verlag: Berlin, Germany, 2000.
55. The Object Management Group (OMG). See `http://www.omg.org/`.
56. Andrea Omicini. Soda: Societies and infrastructures in the analysis and design of agent-based systems. In P. Ciancarini and M. Wooldridge, editors, *Agent-Oriented Software Engineering — Proceedings of the First International Workshop (AOSE-2000)*. Springer-Verlag: Berlin, Germany, 2000.

57. A. Pnueli. Specification and development of reactive systems. In *Information Processing 86*. Elsevier Science Publishers B.V.: Amsterdam, The Netherlands, 1986.
58. 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.
59. 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.
60. 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.
61. 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.
62. 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.
63. 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.
64. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliifs, NJ, 1991.
65. Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, 60(1):51–92, 1993.
66. R. G. Smith. *A Framework for Distributed Problem Solving*. UMI Research Press, 1980.
67. R. M. Smullyan. *First-Order Logic*. Springer-Verlag: Berlin, Germany, 1968.
68. M. Spivey. *The Z Notation (second edition)*. Prentice Hall International: Hemel Hempstead, England, 1992.
69. G. Weiß, editor. *Multi-Agent Systems*. The MIT Press: Cambridge, MA, 1999.
70. Mark Wood and Scott A. DeLoach. An overview of the multiagent systems engineering methodology. In P. Ciancarini and M. Wooldridge, editors, *Agent-Oriented Software Engineering — Proceedings of the First International Workshop (AOSE-2000)*. Springer-Verlag: Berlin, Germany, 2000.
71. M. Wooldridge. *The Logical Modelling of Computational Multi-Agent Systems*. PhD thesis, Department of Computation, UMIST, Manchester, UK, October 1992.
72. 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: Berlin, Germany, January 1995.
73. M. Wooldridge. Agent-based software engineering. *IEE Proceedings on Software Engineering*, 144(1):26–37, February 1997.
74. M. Wooldridge. *Reasoning about Rational Agents*. The MIT Press: Cambridge, MA, 2000.
75. M. Wooldridge and N. R. Jennings. Intelligent agents: Theory and practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.
76. 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.
77. M. Wooldridge, N. R. Jennings, and D. Kinny. A methodology for agent-oriented analysis and design. In *Proceedings of the Third International Conference on Autonomous Agents (Agents 99)*, pages 69–76, Seattle, WA, May 1999.