On the Sources of Complexity in Agent Design

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

To appear in *Applied Artificial Intelligence*.

Abstract

This article addresses the question of what makes agent design hard. This is a fundamental issue not just for the intelligent agents community, but also for the computer science and software engineering communities in general. The article begins by presenting a simple, informal model of agents and their environments, and then uses this model to frame a discussion on the sources of complexity in agent design. Three key sources of complexity are identified: the agent's environment, the nature of interaction between agent and environment, and the nature of the specification, or task for the agent. Following this discussion, the article presents a simple formal model of agents and their environments, which is capable of representing many of these sources of complexity. Using the formal model, a number of stereotypical agent design problems are defined. The article concludes with a discussion of related work and issues.

1 Introduction

Today's computer systems are the most complex mass market products ever developed. A typical operating system for a personal computer, together with applications, contains of the order of 10^8 lines of program code. Unfortunately, it has long been recognised that software has properties that make it inherently difficult to correctly and efficiently engineer [Brooks, 1975]. The discipline of software engineering is about understanding and managing this complexity.

This article is all about complexity — the complexity that we encounter when engineering computer systems. Such complexity evidences itself in two very different ways: *software engineering* complexity, in the sense of *The Mythical Man Month* [Brooks, 1975], and *computational complexity*, in the sense of NP-completeness and Cook's theorem [Papadimitriou, 1994].

As its name suggests, software engineering complexity relates to the difficulty of designing and successfully implementing a particular computer system. There is (as yet) no science of software engineering complexity — our understanding of what makes one particular system more difficult to engineer than another is to date based on not much more than anecdote.

Computational complexity is primarily concerned with determining precise upper and lower bounds on the amount of computation time and memory space required to solve particular problems, and on developing efficient algorithms to solve these problems. Great progress has been made over the past three decades in classifying problems according to their space and time complexity, and of understanding the relationships between different complexity classes [Papadimitriou, 1994].

This article considers the *intelligent agent design problem* from these two perspectives. Intelligent agents are computer systems that are capable of independent action in some environment in order to achieve their design objectives [Wooldridge and Jennings, 1995]. Agents have recently been promoted as a promising new paradigm for software development [Wooldridge, 1997, Jennings, 1999]. The basic question addressed by the paper is: *What makes agent development hard?* The paper begins by discussing the software engineering complexity of the agent design problem. It is proposed that three key sources of such complexity are *the nature of the environment* in which the agent is situated, *the nature of the interaction* the agent has with its environment, and finally, *the nature of the task* the agent is intended to accomplish. Following this discussion, a simple formal model of agents is presented. This model is used to formally define some stereotypical agent design problems. The article closes with a discussion of related work and some conclusions and issues.

Note that this article is intended as a survey paper, which should be accessible to readers both from the software engineering and artificial intelligence communities. In particular, it builds upon the discussions in [Russell, 1997] and [Wooldridge, 1997].

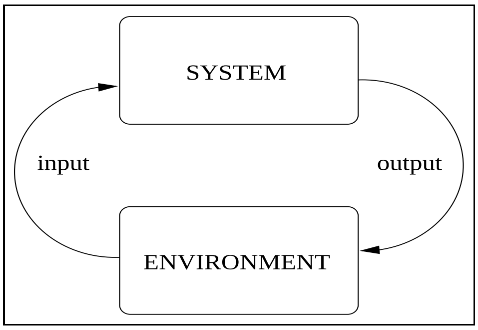

Figure 1: An abstract view of software systems.

2 Complexity in Agent Design

In order to understand what makes certain types of software more complex than others, we 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 that produces output that affects the environment. The environment is often software (e.g., an operating system such as UNIX or Windows 98), so actions take the form of software operations such as writing to a file. However, if the system is embedded within a physical environment, then actions performed by the agent will correspond to "real world" actions, such as picking up and moving objects. One aspect of software that we cannot capture in Figure 1 is the fact that we build software for some purpose. We start from a *specification* of what we want; this specification will then allow us to determine whether or not a particular system does or does not meet our requirements.

Given this abstract view, we can examine the factors that affect the complexity of engineering such 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 [Russell and Norvig, 1995].

- *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. 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.

- *The nature of the task to be solved.*

  The task to be solved is defined by a 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 view of specifications that is implicit within the formal methods community, as typified by work on the Z and VDM specification languages. 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*.

In the sub-sections that follow, we consider these sources of complexity in more detail.

2.1 The Nature of the Environment

The most obvious source of complexity in agent design is the agent's environment. In [Russell and Norvig, 1995, p46], the following sources of environmental complexity are identified:

- *Accessible* vs *inaccessible.*

  An accessible environment is one in which the agent can obtain complete, accurate, up-to-date information about the environment's state. Most real-world environments (including, for example, the everyday physical world and the Internet) are not accessible in this sense.

- *Deterministic* vs *non-deterministic.*

  A deterministic environment is one in which any action has a single guaranteed effect — there is no uncertainty about the state that will result from performing an action.

- *Static* vs *dynamic.*

A static environment is one that can be assumed to remain unchanged except by the performance of actions by the agent. In contrast, a dynamic environment is one that has other processes operating on it, and which hence changes in ways beyond the agent's control. The physical world is a highly dynamic environment, as is the Internet.

- *Discrete* vs *continuous.*

  An environment is discrete if there are a fixed, finite number of actions and percepts in it.

We begin our discussion with accessibility. First, note that in extreme cases, the laws of physics prevent many environments from being completely accessible. For example, it may be that as I write, the surface temperature at the North Pole of Mars is -100 degrees Celsius, but the laws of physics will prevent me from knowing this fact for some time. This information is thus *inaccessible* to me. More mundanely, in almost any realistic environment uncertainty is inherently present.

The more accessible an environment is, the simpler it is to build agents that operate effectively within it. The reason for this should be self-evident. Ultimately, a "good" agent is one that makes the "right" decisions. The quality of decisions that an agent can make is clearly dependent on the quality of the information available to it. If little, or inaccurate information is available, then the agent's decision is uninformed, and is hence likely to be poor. As more complete and accurate information becomes available, the potential to make a good decision increases.

The next source of complexity we consider is *determinism.* An environment is deterministic if the outcome of any action performed is uniquely defined, and non-deterministic otherwise. Non-determinism can seem an unusual property to attribute to environments. For example, we usually imagine that software environments, governed as they are by precise rules, are paradigms of determinism. Non-determinism captures several important aspects of such environments:

- Non-determinism captures the fact that agents have a limited "sphere of influence" — they have at best partial control over their environment.

- Similarly, actions are typically performed by agents in order to bring about some desired state of affairs. Non-determinism captures the fact that actions can *fail* to have the desired result.

Clearly, deterministic environments are preferable from the point of view of the agent designer to non-deterministic environments. If there is never any uncertainty about the outcome of some particular action, then an agent need never stop to determine whether or not a particular action had a particular outcome, and thus whether or not it needs to reconsider its course of action. In particular, in a deterministic environment, an agent designer can assume that the actions performed by an agent will always succeed: they will never fail to bring about their intended effect.

Unfortunately, as [Russell and Norvig, 1995] point out, if an environment is sufficiently complex, then the fact that it is *actually* deterministic is not much help. To all intents and purposes, it may as well be non-deterministic. In practice, almost all realistic environments must be regarded as non-deterministic from an agent's perspective.

Non-determinism is closely related to *dynamism.* Early artificial intelligence research on action selection focussed on planning algorithms — algorithms that, given a description of the initial state of the environment, the actions available to an agent and their effects, and a goal state, will generate a plan (i.e., a sequence of actions) such that when executed from the initial environment state, the plan will guarantee the achievement of the goal [Allen et al., 1990]. However, such planning algorithms implicitly assumed that the environment in which the plan was being executed was *static* — that it did not change except through the performance of actions by the agent. Clearly, many environments (including software environments such as computer operating systems, as well as physical environments such as the real world), do not enjoy this property — they are dynamic, with many processes operating concurrently to modify the environment in ways that an agent has no control over.

From an agent's point of view, dynamic environments have at least two important properties. The first is that if an agent performs no external action between times t_0 and t_1, then it cannot assume that the environment at t_1 will be the same as it was at time t_0. This means that in order for the agent to select an appropriate action to perform, it must perform *information gathering* actions to determine the state of the environment [Moore, 1990]. In a static environment, there is no need for such actions. The second property is that other processes in the environment can "interfere" with the actions it attempts to perform. The idea is essentially the concept of interference in concurrent systems theory [Ben-Ari, 1990]. Thus if an agent checks that the environment has some property φ and then starts executing some action α on the basis of this information, it cannot in general guarantee that the environment will continue to have property φ while it is executing α.

These properties imply that static environments will be inherently simpler to design agents for than dynamic ones. First, in a static environment, an agent need only ever perform information gathering actions *once*. Assuming the information it gathers correctly describes the environment, and that it correctly understands the effects of its actions, then it can accurately *predict* the effects of its actions on the environment, and hence how the state of the environment will evolve. (This is in fact how most artificial intelligence planning algorithms work [Lifschitz, 1986].) Second, in a static environment an agent need never worry about *synchronizing* or *coordinating* its actions with those of other processes in the environment [Bond and Gasser, 1988].

The final distinction made in [Russell and Norvig, 1995] is between *discrete* and *continuous* environments. A discrete environment is one that can be guaranteed to only ever be in a finite number of discrete states; a continuous one may be in uncountably many states. Thus the game of chess is a discrete environment — there are only a finite (albeit very large) number of states of a chess game. [Russell and Norvig, 1995] give taxi driving as an example of a continuous environment.

Discrete environments are simpler to design agents for than continuous ones, for several reasons. Most obviously, digital computers are themselves discrete-state systems, and although they can simulate continuous systems to any desired degree of accuracy, there is inevitably a mismatch between the two types of systems. Some information must be lost in the mapping from continuous environment to discrete representation of that environment. Thus the information a discrete-state agent uses in order to select an action in a continuous environment will be made on the basis of information that is inherently approximate. Finally, with finite discrete state environments, it is in principle possible to enumerate all possible states of the environment and the optimal action to perform in each of these states. Such a lookup table approach to agent design is rarely possible in practice, but it is at least *in principle* possible for finite, discrete state environments.

In summary, the most complex general class of environments are those that are inaccessible, non-deterministic, dynamic, and continuous. Environments that have these properties are often referred to as *open* [Hewitt, 1986].

2.2 The Interaction between Agent and Environment

Environmental properties have a role in determining the complexity of the agent design process, but they are by no means the only factors that play a part. The second important property that plays a part is the nature of the *interaction* between agent and environment.

Originally, software engineering concerned itself with what are known as "functional" systems. A functional system is one that simply takes some input, performs some computation over this input, and eventually produces some output. Such systems may formally be viewed as functions $f : I \rightarrow O$ from a set I of inputs to a set O of outputs. The classic example of such a system is a compiler, which can be viewed as a mapping from a set I of legal source programs to a set O of corresponding object or machine code programs.

One of the key attributes of such functional systems is that they *terminate*. This means that, formally, their properties can be understood in terms of pre- and post-conditions [Hoare, 1969]. The idea is that a pre-condition φ represents what must be true of the program's environment in order for that program to operate correctly. A post-condition ψ represents what will be true of the program's environment after the program terminates, assuming that the pre-condition was satisfied when execution of the program commenced. A program is said to be completely correct with respect to pre-condition φ and post-condition ψ if it is guaranteed to terminate when it is executed from a state where the pre-condition is satisfied, and upon termination, its post-condition is guaranteed to be satisfied. Crucially, it is assumed that the agent's environment, as characterised by its pre-condition φ, is *only* modified through the actions of the program itself. As we noted above, this assumption does not hold for many environments.

Although the internal complexity of a functional system may be great (e.g., in the case of a compiler for a complex programming language such as Ada), functional programs are, in general, comparatively simple to correctly and efficiently engineer. For example, functional systems lend themselves to design methods based on "divide and conquer" — top-down stepwise refinement [Jones, 1990] is an example of such a method. Semi-automatic refinement techniques are also available, that allow a designer to refine a high-level (formal) specification of a functional system down to an implementation [Morgan, 1994].

Unfortunately, many computer systems that we desire to build are not functional in this sense. Rather than simply computing a function of some input and then terminating, many computer systems are *reactive*, in the following sense[1]:

> "Reactive systems are systems that cannot adequately be described by the *relational* or *functional* view. The relational view regards programs as functions ... from an initial state to a terminal state. Typically, the main role of reactive systems is to maintain an interaction with their environment, and therefore must be described (and specified) in terms of their on-going behaviour ... [E]very concurrent system ... must be studied by behavioural means. This is because each individual module in a concurrent system is a reactive subsystem, interacting with its own environment which consists of the other modules". [Pnueli, 1986]

Reactive systems are harder to engineer than functional ones. Perhaps the most important reason for this is that an agent engaging in a (conceptually) non-terminating relationship with its environment must continually make *local* decisions that have *global* consequences. Consider a simple printer controller agent. The agent continually receives requests to have access to the printer, and is allowed to grant access to any agent that requests it, with the proviso that it is only allowed to grant access to one agent at a time. At some time, the agent reasons that it will give control of the printer to process p_1, rather than p_2, but that it will grant p_2 access at some later time point. This seems like a reasonable decision, when considered in isolation. But if the agent *always* reasons like this, it will *never* grant p_2 access[2]. In other words, a decision that seems entirely reasonable in a local context can have undesirable effects when considered in the context of the system's entire history. This is a simple example of a complex problem. In general, the decisions made by an agent have long-term effects, and it is often difficult to understand such long-term effects.

[1]There are at least three current usages of the term *reactive system* in computer science. The first, oldest, usage is that by Pnueli and followers (see, e.g., [Pnueli, 1986], and the description above). Second, researchers in AI planning take a reactive system to be one that is capable of responding rapidly to changes in its environment — here the word 'reactive' is taken to be synonymous with 'responsive' (see, e.g., [Kaelbling, 1986]). More recently, the term has been used to denote systems which respond directly to the world, rather than reason explicitly about it (see, e.g., [Connah and Wavish, 1990]). In this paper the term is used in its Pnuelian sense, except where otherwise indicated.

[2]In mainstream computer science, this is known as *fairness* [Francez, 1986].

One possible solution is to have the agent explicitly reason about and predict the behaviour of the system, and thus any temporally distant effects, at run-time. But it turns out that such prediction is extremely hard — representing and reasoning about the behaviour of most moderately sized systems is simply beyond the capabilities of today's knowledge representation and reasoning techniques. Of course, such prediction need not be done at run-time. An alternative is for prediction to be done off-line, by a designer. Whatever solution is adopted, the presence of non-local effects clearly makes the agent design process much more complex than it would otherwise be.

[Russell and Subramanian, 1995] discuss the essentially identical concept of *episodic* environments. In an episodic environment, the performance of an agent is dependent on a number of discrete episodes, with no link between the performance of the agent in different episodes. An example of an episodic environment would be a mail sorting system [Russell and Subramanian, 1995]. As with reactive systems, episodic interactions are simpler from the agent developer's perspective because the agent can decide what action to perform based only on the current episode — it need not reason about the interactions between this and future episodes.

Another aspect of the interaction between agent and environment is the concept of *real time*. Put at its most abstract, a real-time interaction is simply one in which time plays a part in the evaluation of an agent's performance [Russell and Subramanian, 1995, p585]. It is possible to identify several different types of real-time interactions:

- those in which a decision must be made about what action to perform within some specified time bound;
- those in which the agent must bring about some state of affairs as quickly as possible;
- those in which an agent is required to repeat some task, with the objective being to repeat the task as often as possible.

If time is not an issue, then an agent can deliberate for as long as required in order to select the "best" course of action in any given scenario. Selecting the best course of action implies search over the space of all possible courses of action, in order to find the "best". Selecting the best action in this way will take time exponential in the number of actions available to the agent[3]. It goes without saying that for any realistic environment, such deliberation is not viable. Thus any realistic system must be regarded as real-time in some sense.

Some environments are real-time in a much stronger sense than this. For example, the PRS, one of the best-known agent systems, had fault diagnosis on NASA's space shuttle as its initial application domain [Georgeff and Lansky, 1987]. In order to be of any use, decisions in such a system must be made in milliseconds.

2.3 The Nature of the Task

The third aspect of complexity relates to the nature of the task the agent is intended to achieve. The task an agent is intended to achieve is defined by its *specification*. In mainstream computer science, specifications for programs are usually regarded as *predicates*. For example, when temporal logic is used for formal system specification, the specification is expressed a temporal logic formula that is interpreted over program execution histories [Manna and Pnueli, 1992], [Manna and Pnueli, 1995]. This formula is in effect a *predicate* over such histories, which distinguishes between "acceptable" histories (those that satisfy the formula), and "unacceptable" histories (those that do not satisfy the formula). If a program is *guaranteed* to produce a history that satisfies the specification formula, then we say the program satisfies the specification, i.e., it is correct with respect to the specification. Otherwise, it does not satisfy the specification. Such specifications induce an equivalence relation over programs, whereby all programs that satisfy the specification are considered equally good, and all those that fail to satisfy it are considered equally bad.

Specifications as predicates are a special case of a more general type of specification, whereby we associate with each possible history some numeric value, called its *utility*. We prefer history h_1 to history h_2 if the utility of h_1 is greater than h_2. The specification for such systems is given by a *utility function*, which assigns real numbers — utilities — to histories. The aim when implementing such a specification is to *maximise* utility — there is thus no sense of an agent either satisfying or not satisfying a specification. We tend to say that an agent is *rational* if it acts to maximise utility. A direct example of a utility maximising specification would be an agent for trading stocks and shares on the World-Wide

[3] If the agent has n actions available to it, then it has $n!$ courses of action available to it (assuming no duplicate actions).

Web, in which case utility equates directly to money earned. But utility is a much more general concept than simply cash earned. For example, the utility for a "factory" agent might be number of widgets manufactured.

Many tasks that we intuitively consider "difficult" are of the utility maximising type, rather than the predicate type. For example, a chess playing program aims to maximise its performance against human players. A predicate-based specification for a chess playing system would therefore be too simplistic. An interesting way of distinguishing artificial intelligence from mainstream computer science would be to say that mainstream computer science is concerned with systems specified by predicates, whereas artificial intelligence is concerned with utility maximising systems. [Russell and Norvig, 1995, p472] expresses a similar view.

We stated above that specifications as predicates and specifications as utility functions were related. To see how, suppose that we had some utility function that was guaranteed to assign a program history one of only two values. For simplicity, we assume these values are normalised to 0 (read as "false") and 1 (read as "true"). Then the utility function induces exactly the same equivalence relation over programs that predicates do: either a program satisfies the specification (is assigned the value 1 — "true"), or it doesn't (is assigned the value 0 — "false"). A program is then viewed as *rational* to the extent that it satisfies its specification.

3 A Formal Model

In this section, we present a simple formal model of agents and the environments they occupy, which allows us to capture many of the sources of complexity discussed above. An overview of this framework is given in Figure 2 (cf. [Genesereth and Nilsson, 1987]). We start by introducing the basic sets used in our formal model. First, it is assumed that the environment may be in any of a set

$$E = \{e, e', \ldots\}$$

of states. We also assume the (single) agent occupying this environment may be in any of a set

$$L = \{l, l', \ldots\}$$

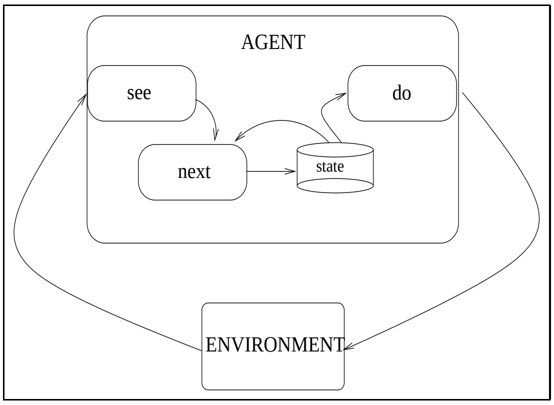

Figure 2: An overview of the framework.

of *local* states. Agents are assumed to have a repertoire of possible actions available to them, which transform the state of the environment. We let

$$Ac = \{\alpha, \alpha', \ldots\}$$

be the set of actions. We assume a distinguished member ϵ of Ac, representing the "noop" action, which has no effect on the environment.

3.1 Environments

In order to represent the effect that an agent's actions have on an environment, we introduce a *state transformer* function (cf. [Fagin et al., 1995, p154]),

$$\tau : E \times Ac \to \wp(E).$$

Thus $\tau(e, \alpha)$ denotes the possible environment states that could result from performing action α in environment state e. Note that this definition allows for non-determinism in the environment. There is *uncertainty* about the result of performing an action in some state. If $\tau(e, \alpha)$ is a singleton, for all $e \in E$ and $\alpha \in Ac$, then the environment is deterministic.

We noted above that some environments are *inaccessible* — it is not possible to know everything about the environment. In order to represent the fact that there is uncertainty in an environment, we use a *visibility function*:

$$\nu : E \to (\wp(E) \setminus \emptyset)$$

The idea is that if the environment is *actually* in state e, then it is impossible for the agent to distinguish between e and any member of $\nu(e)$. We will require that ν partitions E into mutually disjoint sets of states, and that $e \in \nu(e)$, for all $e \in E$. (The idea of a visibility function is related to the concept of indistiguishability in epistemic logic, though we shall not here investigate the relationship [Fagin et al., 1995].)

We say ν is *accessible* if $\nu(e) = \{e\}$. Intuitively, if ν is accessible, then it is possible for an agent observing the environment to identify exactly what state it is in. In other words, an agent equipped with suitable sensory apparatus could learn everything there is to learn about an accessible environment. Similarly, we say that ν is *inaccessible* if $\nu(e) = E$. If an environment is inaccessible, then this implies it carries the minimum possible information. Most environments are neither entirely accessible nor entirely inaccessible, but lie somewhere between the two extremes.

Formally, an environment Env is a 4-tuple

$$Env = \langle E, \tau, \nu, e_o \rangle$$

where E is a set of environment states as above, τ is a state transformer function, ν is a visibility function, and $e_0 \in E$ is the initial state of Env.

3.2 Agents

From Figure 2, we can see that an agent has three functional components, representing its sensors (the function *see*), its next state function (*next*), and its action selection, or decision making function (*do*). Formally, the perception function

$$see : S \to P$$

maps environment states to *percepts* — we denote members of P by $\rho, \rho', \ldots$. The agent's next state function

$$next : L \times P \to L$$

maps an internal state and percept to an internal state. Finally, the action selection function

$$do : L \to Ac$$

simply maps internal states to actions.

The behaviour of an agent can be summarised as follows. The agent starts in some state l_0. It then observes its environment state e through the visibility function $\nu(e)$, and generates a percept $see(\nu(e))$. The internal state of the agent is then updated, and becomes $next(l_0, see(\nu(s)))$. The action selected by the agent is then $do(next(l_0, see(\nu(s))))$. This action is performed, and the agent enters another cycle.

Together, an environment/agent pair comprise a *system*. The *global* state of a system at any time is a pair containing the state of the agent and the state of the environment. Let

$$G = E \times L$$

be the set of all such global states. We use g (with annotations: $g, g', \ldots$) to stand for members of G. A *run* r of a system can thus be thought of as an infinite sequence:

$$r : g_0 \xrightarrow{\alpha_0} g_1 \xrightarrow{\alpha_1} g_2 \xrightarrow{\alpha_2} g_3 \xrightarrow{\alpha_3} \cdots \xrightarrow{\alpha_{u-1}} g_u \xrightarrow{\alpha_u} \cdots$$

Formally, an infinite sequence

$$(g_0, g_1, g_2, \ldots)$$

over G represents a run of an agent $Ag = \langle see, next, do, l_0 \rangle$ in an environment $Env = \langle E, \tau, \nu, e_0 \rangle$ iff $g_0 = \langle e_0, l_0 \rangle$ and for all u in $I\!N$, we have

$$g_{u+1} = \langle e_{u+1}, next(l_u, see(\nu(e_u))) \rangle.$$

where $e_{u+1} \in \tau(e_u, action(l_u))$. In other words, the state of the external environment at time $u+1$ is determined through the τ function by the state of the environment at time u and the action performed by the agent at time u. The state of the agent at time $u+1$ is determined through the *next* function by the agent's internal state at time u and the percept the agent received by observing the environment at u.

Let $runs(Ag, Env)$ denote the set of all runs of an agent Ag in environment Env, and let Run be the set of all runs.

3.3 Task Environments

As we noted above, we want our agents to achieve tasks for us. In order to measure how well they do this, we introduce the *value*, or *utility* of a run via a function

$$V : Run \to \mathbb{R}$$

which assigns real numbers denoting utilities, or payoffs to runs. Note that runs are infinite sequences, isomorphic to the natural numbers, and so V represents the utility of a run at infinity. Thus V does not capture the standard decision-theoretic notion of utility, where values are assigned to *states*. We will assume that there is some upper bound to the values that V assigns, so there will always be at least one "optimal" run.

A *task environment* is defined to be a pair $\langle Env, V \rangle$, where Env is an environment, and $V : Run \to \mathbb{R}$ is a utility function. We define the utility of an agent with respect to a task environment $\langle Env, V \rangle$ to be the minimum possible utility that could be obtained by Ag in $\langle Env, V \rangle$. We will abuse notation slightly and write $V(Ag, Env)$ to stand for the utility obtained by Ag in task environment $\langle Env, V \rangle$:

$$V(Ag, Env) = \min\{V(r) \mid r \in runs(Ag, Env)\}.$$

Thus $V(Ag, Env)$ is a pessimistic assessment of the utility of an agent in a particular environment — if $V(Ag, Env) = k$ then agent Ag is guaranteed to obtain *at least* payoff $k \in \mathbb{R}$.

We say an agent Ag is *optimal* with respect to a task environment $\langle Env, V \rangle$ if there is no agent Ag' such that $V(Ag', Env) > V(Ag, Env)$ (cf. the concept of an optimal agent as one that maximises expected utility [Russell and Subramanian, 1995, p583]). There will always be at least one optimal agent for any given task environment. An agent Ag_{opt} that is optimal for some task environment $\langle Env, V \rangle$ satisfies the following equation.

$$Ag_{opt} = \arg\max_{Ag} V(Ag, Env)$$

Note that we could easily extend the state transformer function τ to include a probability distribution over possible outcomes, and hence induce a probability distribution over runs. This would allow us to define $V(Ag, Env)$ as the expected payoff of agent Ag in environment V [Russell and Subramanian, 1995, p583].

We have now defined a simple formal model of agents and their environments, and given a precise meaning to the concept of agent design. In the sections that follow, we see how this model can be used to capture many of the sources of complexity that we discussed above. First, note that:

- the model allows for environments of arbitrary accessibility, via the accessibility function ν;
- non-determinism is explicit within the state transformer function τ;
- the properties of arbitrary dynamic environments may be captured via non-determinism in τ;
- continuous environments can be captured if we allow E to be uncountably infinite;
- both reactive and functional systems can be captured in the model of task environments;
- the role of time in evaluating an agent's performance can be captured in the utility function V; and finally,
- the distinction between predicate and utility-based specifications can be captured in the function V.

4 Agent Design

The agent design problem can now be stated as follows:

> Given a task environment $\langle Env, V \rangle$, construct an optimal agent.

Expressed as a satisfiability-like decision problem [Papadimitriou, 1994], the agent design problem becomes:

> Given a task environment $\langle Env, V \rangle$, answer "yes" if there exists an agent Ag_{sat} such that $V(Ag_{sat}) \geq k$ for some constant $k \in \mathbb{R}$; answer "no" otherwise.

Here k represents a "satisfactory" value; we would be happy with any agent that achieved at least k.

One obvious problem with defining agent design problem in this way is that an optimal agent is assumed to be the best agent that is possible *in principle*: the optimal agent is obtained from the set of all possible agents. But this is not in general the set of agents that we should be considering. Instead, we should be focussing on the set of agents that can be *implemented* on the computing devices that we have at hand. For example, there is not much point in knowing that a particular agent Ag_{2485} is optimal for our task environment if there is no way of implementing Ag_{2485} on any computer currently available. This consideration motivates the definition of *bounded optimal* agents [Russell and Subramanian, 1995].

Let $M = \{m, m', \ldots\}$ be the set of all possible "machines". These machines are the computational devices upon which we will implement our agents. Further, let us write $implementable(Ag, m)$ to denote the fact that agent Ag can be implemented on machine $m \in M$. Then given task environment $\langle Env, V \rangle$ and machine m, an optimal agent Ag_{opt} satisfies the following equation.

$$Ag_{opt} = \arg \max_{Ag \in \{Ag' \mid implementable(Ag', m)\}} V(Ag, Env)$$

Defining Ag_{opt} in this way ensures that we will actually be able to implement it — although, of course, it says nothing about how we *obtain* f_{opt}.

4.1 Types of Tasks

In practice, it is rare for a task to be specified directly in terms of a utility function. Instead, tasks are usually specified in one of the following three forms:

1. "achieve state of affairs p";

2. "maintain state of affairs q";

3. some combination of (1) and (2).

We refer to tasks of the first type as *achievement tasks*, and tasks of the second type as *maintenance tasks*. As it turns out, all three of these types of tasks can be defined as predicate based specifications.

Consider first achievement tasks, such as the Towers of Hanoi, or the blocks world problem. In such problems, the objective of an agent is to *achieve some* *state of affairs.* We refer to task environments that define such problems as *achievement-based.* More precisely, the predicate task environment $\langle Ag, Env \rangle$ is achievement-based if it accords an agent the utility 1 iff that agent is guaranteed to bring about any one of a set of goal states $E_{good} \subseteq E$. Formally, the predicate task environment $\langle Env, V \rangle$ is achievement-based iff $\forall r \in runs(Ag, Env)$, we have $V(r) = 1$ iff $\exists u \in I\!N$ such that $r(u) \in E_{good}$ for some $E_{good} \subseteq E$. Given an achievement-based task environment, there is an obvious decision problem:

> [Achievement-based implementation.]
> Given the achievement-based task environment $\langle Env, V \rangle$ with goal $E_{good} \subseteq E$, does there exist an agent Ag such that $\forall r \in runs(Ag, Env)$, $\exists u \in I\!N$ such that $e(r(u)) \in E_{good}$?

A useful way to think about this problem is as the agent as *playing a game* against the environment. In the terminology of game theory [Binmore, 1992], this is exactly what is meant by a "game against nature". The environment and agent both begin in some state; the agent takes a turn by executing an action, and the environment responds with some state; the agent then takes another turn, and so on. The agent "wins" if it can force the environment into one of the goal states E_{good}. The achievement-based implementation problem can then be understood as asking whether or not there is a *winning strategy* that can be played against $\langle Env, V \rangle$ to bring about one of E_{good}. This class of problem — determining whether or not there is a winning strategy for one player in a particular two-player game — is closely associated with PSPACE-complete problems [Papadimitriou, 1994]. Note that it is generally believed that PSPACE-complete problems are more complex than NP-complete problems.

Just as many predicate task environments can be characterised as problems where an agent is required to bring about some state of affairs, so many others can be classified as problems where the agent is required to *avoid* some state of affairs. As an extreme example, consider a nuclear reactor agent, the purpose of which is to ensure that the reactor never enters a "meltdown" state. Somewhat more mundanely, we can imagine a software agent, one of the tasks of which is to ensure that a particular file is never simultaneously open for both reading and writing. We call such task environments *maintenance-based.* The predicate task environment $\langle Ag, Env \rangle$ is maintenance-based if it accords an agent the utility 1 iff that agent is guaranteed *never* to bring about any state $E_{bad} \subseteq E$. Formally, the predicate task environment $\langle Env, V \rangle$ is maintenance-based iff $\forall r \in runs(Ag, Env)$, we have $V(r) = 1$ iff $\forall u \in I\!N$, $e(r(u)) \notin E_{bad}$ for some $E_{bad} \subseteq E$. As with achievement-based task environments, there is an obvious decision problem.

> [The maintenance-based implementation problem.]
> Given a maintenance-based task environment $\langle Env, V \rangle$ with states $E_{bad} \in E$, does there exist an agent Ag such that $\forall r \in runs(Ag, Env)$, $\forall u \in I\!N$ we have $e(r(u)) \notin E_{bad}$?

It is again useful to think of the problem as a game. This time, the agent wins if it manages to *avoid* E_{bad}. The environment, in the role of opponent, is attempting to force the agent into E_{bad}; the agent is satisfactory if it has a winning strategy for avoiding E_{bad}.

Many real-world problems *combine* elements of both achievement and maintenance-based task environments. In particular, their properties can be classified in terms of both *liveness* and *safety* properties [Pnueli, 1986]. Liveness properties resemble achievement problems. Intuitively, they state that "something good happens". For example, a typical liveness property would state that "if a request is received, then a response will eventually be sent". Safety properties, in contrast, are identical to our maintenance properties — they define a set of states that the system is required never to enter.

It would be hard to specify a complex set of inter-related liveness and safety properties directly via the function V. In particular, it is necessary to specify the desired behaviour of the system over an *infinite* sequence of states. An obvious question is therefore how to specify the combination of properties. Temporal logic has proven to be useful for this purpose [Pnueli, 1986].

5 Related Work

The relationship between artificially intelligent agents and software complexity has been discussed by several researchers; [Simon, 1981] was probably the first. More recently, [Booch, 1994] gives a good discussion of software complexity and the role that object-oriented development has to play in overcoming it. [Russell and Norvig, 1995] introduced the five-point classification of environments that we reviewed here, and distinguished between the "easy" and "hard" cases. [Kaelbling, 1986] touches on many of the issues discussed here, and [Jennings, 1999] also discusses the issues associated with complexity and agents.

The formal model of agents and environments is similar to many that are now used in artificial intelligence: [Fagin et al., 1995] and [Genesereth and Nilsson, 1987] are two examples. [Russell and Subramanian, 1995] made a distinction between *optimal* agents (which are the best agents that exist in *theory* for a given task environment) and *bounded optimal* agents (which are the best one can implement *in practice*, on a given computer).

The relationship between agent and environment, and in particular, the problem of understanding how a given agent will perform in a given environment, has been studied empirically by several researchers. [Pollack and Ringuette, 1990] introduced the TILEWORLD, an environment for experimentally evaluating agents that allowed a user to experiment with various environmental parameters (such as the rate at which the environment changes — its *dynamism*). Building on this work, [Kinny and Georgeff, 1991] investigated how a specific class of agents, based on the *belief-desire-intention* model, could be tailored to perform well in environments with differing degrees of change and complexity. An attempt to prove some results corresponding to [Kinny and Georgeff, 1991] was [Wooldridge and Parsons, 1999]. An informal discussion on the relationship between agent and environment is [Müller, 1999].

Probably the most relevant work from mainstream computer science to that discussed in this paper has been on the application of temporal logic to reasoning about systems [Manna and Pnueli, 1992], [Manna and Pnueli, 1995]. Temporal logic has been particularly applied to the specification of *non-terminating* systems. Temporal logic is particularly appropriate for the specification of such systems because it allows a designer to succinctly express complex properties of infinite sequences of states.

We identified several decision problems for agent design, and closely related problems have also been studied in the computer science literature. Perhaps the closest to our view is the work of Pnueli and Rosner [Pnueli and Rosner, 1989] on the automatic synthesis of reactive systems from branching time temporal logic specifications. They specify a reactive system in terms of a first-order branching time temporal logic formula $\forall x \; \exists y \; A \; \varphi(x, y)$. The predicate φ characterises the relationship between inputs to the system (x) and outputs (y). Inputs may be thought of as sequences of environment states, and outputs as corresponding sequences of actions. The A is a branching time temporal logic connective meaning 'on all paths', or 'in all possible futures'. The specification is intended to express the fact that in all possible futures, the desired relationship φ holds between the inputs to the system, x, and its outputs, y. Pnueli and Rosner show that the time complexity of the synthesis process is doubly exponential in the size of the specification.

In artificial intelligence, the planning problem is most closely related to our achievement-based task environments [Allen et al., 1990]. STRIPS was the archetypal planning system [Fikes and Nilsson, 1971]. The STRIPS system is capable of taking a description of the initial environment state e_0, a specification of the goal to be achieved, E_{good}, and the actions Ac available to an agent, and generates a sequence of actions $\pi \in Ac^*$ such that when executed from e_0, π will achieve one of the states E_{good}. The initial state, goal state, and actions were characterised *declaratively* in STRIPS, using a subset of first-order logic. Bylander showed that the (propositional) STRIPS decision problem (given e_0, Ac, and E_{good} specified in propositional logic, does there exist a $\pi \in Ac^*$ such that π achieves E_{good}?) is PSPACE-complete [Bylander, 1994].

More recently, there has been renewed interest by the artificial intelligence planning community in *decision theoretic* approaches to planning [Blythe, 1999]. One popular approach involves representing agents and their environments as Partially Observable Markov Decision Processes (POMDPs) [Kaelbling et al., 1998]. Put simply, the goal of solving a POMDP is to determine an optimal policy for acting in an environment in which there is uncertainty about the environment state (cf. our visibility function), and which is non-deterministic. Work on POMDP approaches to agent design are at an early stage, but show promise for the future.

Also closely related is the work of Tennenholtz and Moses on the *multi-entity* model of multi-agent systems [Tennenholtz and Moses, 1989]. They use this model to define the *cooperative goal achievement* (CGA) problem, which can be crudely stated as: given a set of benevolent agents, each with their own goals, is there some plan for the set that will achieve all their goals? They show that this problem is PSPACE-complete. This problem is similar in flavour to our achievement-based implementation problem.

6 Conclusions

It seems clear that progress will only be made with agent technology as a software engineering discipline if a greater understanding is obtained of the relationship between agents and their environments [Müller, 1999]. In this article, we have attempted to clarify this relationship. Specifically, we have addressed the issue of what makes the agent design problem hard. We began by presenting a general model of agents and their environments. Using this model, we then informally discussed three of the main sources of complexity: the nature of the environment, the nature of the interaction between agent and environment, and finally, the nature of the task, as defined by the specification. We then presented a formal model of environments, agents, and the tasks they should achieve, and gave a formal definition of the agent design problem. Finally, using this model, we investigated certain special cases of agent design, where the agent is required to achieve some task, and maintain some state of affairs respectively.

A number of problems present themselves as likely candidates for future work. First, the decision problems presented in this paper seem worthy of study in their own right [Papadimitriou, 1994]. A second area of future work is on the use of formal tools for specifying properties of both agents and their environments, that can capture some of the distinctions made in this paper. For example, epistemic logic is known to be a powerful tool for expressing the properties of an environment and the information that an agent carries about the environment [Fagin et al., 1995]; but it is not sufficiently powerful to capture the distinction between what is true of an environment and what may be known about the environment. Finally, much more experience is needed with the use of specific agents for specific types of task in specific environments [Pollack and Ringuette, 1990], [Kinny and Georgeff, 1991]. A finer-grained classification of environment properties is necessary in order to build and deploy successful agents for complex real-world problems.

Acknowledgments

Thanks to Martijn Schutt and Jörg Müller for their comments and careful reading of this paper.

References

[Allen et al., 1990] Allen, J. F., Hendler, J., and Tate, A., editors (1990). *Readings in Planning.* Morgan Kaufmann Publishers: San Mateo, CA.

[Ben-Ari, 1990] Ben-Ari, M. (1990). *Principles of Concurrent and Distributed Programming.* Prentice Hall.

[Binmore, 1992] Binmore, K. (1992). *Fun and Games: A Text on Game Theory.* D. C. Heath and Company: Lexington, MA.

[Blythe, 1999] Blythe, J. (1999). An overview of planning under uncertainty. In Wooldridge, M. and Veloso, M., editors, *Artificial Intelligence Today (LNAI 1600).* Springer-Verlag: Berlin, Germany.

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

[Booch, 1994] Booch, G. (1994). *Object-Oriented Analysis and Design (second edition).* Addison-Wesley: Reading, MA.

[Brooks, 1975] Brooks, F. P. (1975). *The Mythical Man Month.* Addison-Wesley: Reading, MA.

[Bylander, 1994] Bylander, T. (1994). The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69(1-2):165–204.

[Connah and Wavish, 1990] Connah, D. and Wavish, P. (1990). An experiment in cooperation. In Demazeau, Y. and Müller, J.-P., editors, *Decentralized AI — Proceedings of the First European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW-89)*, pages 197–214. Elsevier Science Publishers B.V.: Amsterdam, The Netherlands.

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

[Fikes and Nilsson, 1971] Fikes, R. E. and Nilsson, N. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 5(2):189–208.

[Francez, 1986] Francez, N. (1986). *Fairness.* Springer-Verlag: Berlin, Germany.

[Genesereth and Nilsson, 1987] Genesereth, M. R. and Nilsson, N. (1987). *Logical Foundations of Artificial Intelligence.* Morgan Kaufmann Publishers: San Mateo, CA.

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

[Hewitt, 1986] Hewitt, C. E. (1986). Offices are open systems. *ACM Transactions on Offics Information Systems*, 4(3).

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

[Jennings, 1999] Jennings, N. R. (1999). Agent-based computing: Promise and perils. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI-99)*, pages 1429–1436, Stockholm, Sweden.

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

[Kaelbling, 1986] Kaelbling, L. P. (1986). An architecture for intelligent reactive systems. In Georgeff, M. P. and Lansky, A. L., editors, *Reasoning About Actions & Plans — Proceedings of the 1986 Workshop*, pages 395–410. Morgan Kaufmann Publishers: San Mateo, CA.

[Kaelbling et al., 1998] Kaelbling, L. P., Littman, M. L., and Cassandra, A. R. (1998). Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101:99–134.

[Kinny and Georgeff, 1991] Kinny, D. and Georgeff, M. (1991). Commitment and effectiveness of situated agents. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence (IJCAI-91)*, pages 82–88, Sydney, Australia.

[Lifschitz, 1986] Lifschitz, V. (1986). On the semantics of STRIPS. In Georgeff, M. P. and Lansky, A. L., editors, *Reasoning About Actions & Plans — Proceedings of the 1986 Workshop*, pages 1–10. Morgan Kaufmann Publishers: San Mateo, CA.

[Manna and Pnueli, 1992] Manna, Z. and Pnueli, A. (1992). *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag: Berlin, Germany.

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

[Moore, 1990] Moore, R. C. (1990). A formal theory of knowledge and action. In Allen, J. F., Hendler, J., and Tate, A., editors, *Readings in Planning*, pages 480–519. Morgan Kaufmann Publishers: San Mateo, CA.

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

[Müller, 1999] Müller, J. P. (1999). The right agent (architecture) to do the right thing. In Müller, J. P., Singh, M. P., and Rao, A. S., editors, *Intelligent Agents V — Proceedings of the Fifth International Workshop on Agent Theories, Architectures, and Languages (ATAL-98)*, Lecture Notes in Artificial Intelligence. Springer-Verlag, Heidelberg.

[Papadimitriou, 1994] Papadimitriou, C. H. (1994). *Computational Complexity.* Addison-Wesley: Reading, MA.

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

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

[Pollack and Ringuette, 1990] Pollack, M. E. and Ringuette, M. (1990). Introducing the Tileworld: Experimentally evaluating agent architectures. In *Proceedings of the Eighth National Conference on Artificial Intelligence (AAAI-90)*, pages 183–189, Boston, MA.

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

[Russell and Subramanian, 1995] Russell, S. and Subramanian, D. (1995). Provably bounded-optimal agents. *Journal of AI Research*, 2:575–609.

[Russell, 1997] Russell, S. J. (1997). Rationality and intelligence. *Artificial Intelligence*, 94(1-2):57–78.

[Simon, 1981] Simon, H. A. (1981). *The Sciences of the Artificial (second edition).* The MIT Press: Cambridge, MA.

[Tennenholtz and Moses, 1989] Tennenholtz, M. and Moses, Y. (1989). On cooperation in a multi-entity model: Preliminary report. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence (IJCAI-89)*, Detroit, MI.

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

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

[Wooldridge and Parsons, 1999] Wooldridge, M. J. and Parsons, S. D. (1999). Intention reconsideration reconsidered. In Müller, J. P., Singh, M. P., and Rao, A. S., editors, *Intelligent Agents V — Proceedings of the Fifth International Workshop on Agent Theories, Architectures, and Languages (ATAL-98)*, Lecture Notes in Artificial Intelligence. Springer-Verlag, Heidelberg.