The Computational Complexity of Agent Design Problems

Michael Wooldridge
Department of Computer Science, University of Liverpool
Liverpool L69 7ZF, United Kingdom

M.J.Wooldridge@csc.liv.ac.uk

Abstract

This paper investigates the computational complexity of a fundamental problem in multi-agent systems: given an environment together with a specification of some task, can we construct an agent that will successfully achieve the task in the environment? We refer to this problem as agent design. *Using an abstract formal model of agents and their environments, we begin by investigating various possible ways of specifying tasks for agents, and identify two important classes of such tasks.* Achievement tasks *are those in which an agent is required to bring about one of a specified set of goal states, and* maintenance tasks *are those in which an agent is required to* avoid *some specified set of states. We prove that in the most general case the agent design problem is* PSPACE*-complete for both achievement and maintenance tasks. We briefly discuss the automatic synthesis of agents from task environment specifications, and conclude by discussing related work and presenting some conclusions.*

1 Introduction

There are many approaches to the design of agents that must operate autonomously in some environment [20]. Most of these approaches focus on the problem of *decision making* given fixed time bounds and computational resource constraints. The result has been a range of software architectures for autonomous agents, and an increasing body of work evaluating these architectures in different environmental settings.

To date, the agent community has given comparatively little attention to the basic computational problems that underpin the deployment of agent systems. In this paper, we focus on one particular such problem, which we call *agent design*. The agent design problem arises when we are given a specification of a particular environment, which an agent must inhabit, together with a specification for a task. The agent design problem is simply that of answering whether or not there exists an agent which can be guaranteed to successfully carry out the task in the environment. Of course, an agent that succeeds with a particular task in one environment would not necessarily succeed in another environment. The agent design problem thus involves consideration of both the task *and* environment properties.

We begin our analysis by setting up an abstract model of agents and environments, which we use to formally define the agent design problem. Using this model, section 3 considers the various possible ways in which tasks might be specified. We conclude by identifying two common types of tasks: *achievement tasks* (in which an agent is required to bring about one of a specified set of "goal" states), and *maintenance tasks* (where an agent is required to *avoid* a set of states). We then proceed to analyse the computational complexity of the agent design problem for achievement and maintenance tasks. We prove that, for both types of tasks, the problem is PSPACE-complete in the most general case.

In section 4, we consider a problem that is closely related to agent design: the *automatic synthesis* of agents from task environment specifications. We briefly discuss the implications of our agent design results for synthesis algorithms, focussing particularly on the possibility that tasks are specified *logically*. We discuss related work in AI and computer science in section 5, and present some conclusions in section 6.

Notation: We use standard set theoretic and logical notation wherever possible, augmented as follows. If S is a set, then the set of finite sequences over S is denoted by S^*. If $\sigma \in S^*$ and $s \in S$, then the sequence obtained by appending s to σ is denoted $\sigma \cdot s$. We write $s \in \sigma$ to indicate that element s is present in sequence σ, and write $last(\sigma)$ to denote the final element of σ. Throughout the paper, we assume some familiarity with complexity theory [14].

2 Agents and Environments

In this section, we present an abstract formal model of agents and the environments they occupy; we then use this model to frame the decision problems we study. The systems of interest to us consist of an agent situated in some particular environment; the agent interacts with the environment by performing actions upon it, and the environment responds to these actions with changes in state. It is assumed that the environment may be in any of a finite set $E = \{e, e', \ldots\}$ of instantaneous states. Agents are assumed to have a repertoire of possible actions available to them, which transform the state of the environment. Let $Ac = \{\alpha, \alpha', \ldots\}$ be the (finite) set of actions.

The basic model of agents interacting with their environments is as follows. The environment starts in some state, and the agent begins by choosing an action to perform on that state. As a result of this action, the environment can respond with a number of possible states. However, only one state will *actually* result — though of course, the agent does not know in advance which it will be. On the basis of this second state, the agent again chooses an action to perform. The environment responds with one of a set of possible states, the agent then chooses another action, and so on.

A *run*, r, of an agent in an environment is thus a sequence of interleaved environment states and actions:

$$r : e_0 \xrightarrow{\alpha_0} e_1 \xrightarrow{\alpha_1} e_2 \xrightarrow{\alpha_2} e_3 \xrightarrow{\alpha_3} \cdots \xrightarrow{\alpha_{u-1}} e_u$$

Let $\mathcal{R}$ be the set of all such possible runs. We use $r, r', \ldots$ to stand for members of $\mathcal{R}$.

In order to represent the effect that an agent's actions have on an environment, we introduce a *state transformer* function (cf. [6, p154]):

$$\tau : \mathcal{R} \to 2^E$$

Thus a state transformer function maps a run (assumed to end with the action of an agent) to a set of possible environment states. There are two important points to note about this definition. First, environments are assumed to be *history dependent*. In other words, the next state of an environment is not solely determined by the action performed by the agent and the current state of the environment. The actions made *earlier* by the agent also play a part in determining the current state. Second, note that this definition allows for *non-determinism* in the environment. There is thus *uncertainty* about the result of performing an action in some state.

If $\tau(r) = \emptyset$, (where r is assumed to end with an action), then there are no possible successor states to r. In this case, we say that there are *no allowable actions*, and that the system has *ended* its run. One important assumption we make is that every system is guaranteed to end with runs whose length is polynomial in the size of $Ac \times E$. Thus, we do not consider runs that are infinite in length.

Formally, we say an environment *Env* is a pair $Env = \langle E, \tau \rangle$ where E is a set of environment states, and τ is a state transformer function.

We now need to introduce a model of the agents that inhabit systems. Many architectures for agents have been reported in the literature [20], and one possibility would therefore be to directly use one of these models in our analysis. However, in order to ensure that our results are as general as possible, we choose to model agents simply as functions which map runs (assumed to end with an environment state) to actions (cf. [16, pp580–581]):

$$Ag : \mathcal{R} \to Ac$$

Notice that while environments are implicitly non-deterministic, agents are assumed to be deterministic. Let $\mathcal{AG}$ be the set of all agents.

We say a *system* is a pair containing an agent and an environment. Any system will have associated with it a set of possible runs; we denote the set of runs of agent Ag in environment Env by $\mathcal{R}(Ag, Env)$. Formally, a sequence

$$(e_0, \alpha_0, e_1, \alpha_1, e_2, \ldots)$$

represents a run of an agent Ag in environment $Env = \langle E, \tau \rangle$ iff both:

1. $e_0 = \tau(\epsilon)$ and $\alpha_0 = Ag(e_o)$ (where ϵ is the empty sequence); and
2. for $u > 0$,

$$\begin{array}{lll} e_u & \in & \tau((e_0, \alpha_0, \ldots, \alpha_{u-1})) \quad \text{where} \\ \alpha_u & = & Ag((e_0, \alpha_0, \ldots, e_u)) \end{array}$$

3 Tasks for Agents

We build agents in order to carry out *tasks* for us. The task to be carried out must be *specified* by us. An obvious question is how to specify these tasks. One possibility would be to associate *utilities* with individual states — the task of the agent is then to bring about states that maximise utility. In this approach, a task specification would simply be a function $V : E \to I\!R$, which associated a real value with every environment state. This is similar to the approach used in Markov decision processes [10]. The main disadvantage of this approach is that it assigns utilities to *local* states; it is difficult to specify a *long term* view when assigning utilities to individual states. (Markov decision processes attempt to overcome this problem by "discounting", where states in the short term are considered more important than states in the longer term; this approach has well-known drawbacks.)

Another possibility is to specify a task as a function $V : \mathcal{R} \to I\!R$, which assigns a utility not to individual states, but to runs themselves. If we are concerned with agents that must operate independently over long periods of time, then this approach appears more appropriate to our purposes. However, it is often difficult to define such utility functions.

Rather than attempt to assign real or natural number utilities to runs, we will focus in this paper on a subset of such specifications, where the utility function acts as a *predicate* over runs. Formally, we will say a utility function $V : \mathcal{R} \to I\!R$ is a predicate if the range of V is the set $\{0, 1\}$, that is, if V guarantees to assign a run either 1 ("true") or 0 ("false"). A run $r \in \mathcal{R}$ will be considered to satisfy the specification V if $V(r) = 1$, and fails to satisfy the specification otherwise.

Since we will now focus exclusively on predicate specifications, we will use Ψ to denote a predicate specification, and write $\Psi(r)$ to indicate that run $r \in \mathcal{R}$ satisfies Ψ. In other words, $\Psi(r)$ is true iff $V(r) = 1$. For the moment, we will leave aside the questions of what form a predicate might take (e.g., whether Ψ is expressed in some logical language). Similarly, we will not consider what sorts of tasks might be specified using a predicate specification — we return to these issues below.

We have now introduced the two key components required to frame our decision problems. A *task environment* is defined to be a pair $\langle Env, \Psi\rangle$, where Env is an environment, and $\Psi : \mathcal{R} \to \{0, 1\}$ is a predicate over runs. Let $\mathcal{TE}$ be the set of all task environments. A task environment thus specifies the properties of the system the agent will inhabit (i.e., the environment Env), and also the criteria by which an agent will be judged to have either failed or succeeded (i.e., the specification Ψ).

Given a task environment $\langle Env, \Psi\rangle$, we write $\mathcal{R}_\Psi(Ag, Env)$ to denote the subset of $\mathcal{R}(Ag, Env)$ that satisfy Ψ, that is, $\mathcal{R}_\Psi(Ag, Env) = \{r \mid r \in \mathcal{R}(Ag, Env) \text{ and } \Psi(r)\}$. We then say that an agent Ag succeeds in task environment $\langle Env, \Psi\rangle$ if $\mathcal{R}_\Psi(Ag, Env) = \mathcal{R}(Ag, Env)$. In other words, Ag succeeds in $\langle Env, \Psi\rangle$ if every run of Ag in Env satisfies specification Ψ.

Note that this is in one sense a *pessimistic* definition of success, as an agent is only deemed to succeed if every possible run of the agent in the environment satisfies the specification. If required, we could easily modify the definition of success by extending the state transformer function τ to include a probability distribution over possible outcomes, and hence induce a probability distribution over runs. We can then define the success of an agent as the probability that the specification Ψ is satisfied by the agent. Let $P(r \mid Ag, Env)$ denote the probability that run r occurs if agent Ag is placed in environment Env. Then the probability $P(\Psi \mid Ag, Env)$ that Ψ is satisfied by Ag in Env would then simply be:

$$P(\Psi \mid Ag, Env) = \sum_{r \in \mathcal{R}_\Psi(Ag, Env)} P(r \mid Ag, Env)$$

For the purposes of this paper, however, we will assume that task environments are not stochastic.

We can now express the basic decision problem that we consider throughout the remainder of this paper. We refer to this problem as AGENT DESIGN.

> AGENT DESIGN
> *Given*: task environment $\langle Env, \Psi\rangle$.
> *Answer*: "Yes" if there exists an agent that succeeds in $\langle Env, \Psi\rangle$, "no" otherwise.

The complexity of this problem will in part be determined by the way in which the predicate Ψ is represented. For example, if Ψ is expressed in an undecidable logic, then the corresponding AGENT DESIGN problem will also be undecidable. In what follows, we will consider special cases of the AGENT DESIGN problem, where specification predicates are expressed directly as sets, rather than through the medium of a specification language. In particular, we consider tasks that are specified in one of the following two forms:

1. *Achievement tasks*: "achieve state of affairs φ".
2. *Maintenance tasks*: "maintain state of affairs ψ".

In the subsections that follow, we will discuss both types of tasks in detail.

3.1 Achievement Tasks

Intuitively, an achievement task is specified by a number of goal states; the agent is required to bring about one of these goal states (we do not care which one). Achievement tasks are probably the most commonly studied form of task in artificial intelligence. Many well-known AI problems (e.g., the blocks world) are achievement tasks. A task specified by a predicate Ψ is an achievement task if we can identify some subset $\mathcal{G}$ of environment states E such that $\Psi(r)$ is true just in case one or more of $\mathcal{G}$ occur in r; an agent is successful if it is guaranteed to bring about one of the states $\mathcal{G}$, that is, if every run of the agent in the environment results in one of the states $\mathcal{G}$.

Formally, the task environment $\langle Env, \Psi\rangle$ specifies an achievement task iff there is some $\mathcal{G} \subseteq E$ such that for all $r \in \mathcal{R}(Ag, Env)$, the predicate $\Psi(r)$ is true iff there exists some $e \in \mathcal{G}$ such that $e \in r$. We refer to the set $\mathcal{G}$ of an achievement task environment as the *goal states* of the task; we use $\langle Env, \mathcal{G}\rangle$ to denote the achievement an achievement task environment with goal states $\mathcal{G}$ and environment Env. The decision problem corresponding to AGENT DESIGN for achievement tasks is as follows:

ACHIEVEMENT AGENT DESIGN
Given: achievement task environment $\langle Env, \mathcal{G}\rangle$.
Answer: "yes" if there exists an agent that succeeds in $\langle Env, \mathcal{G}\rangle$, "no" otherwise.

A useful way to think about ACHIEVEMENT AGENT DESIGN is as the agent as *playing a game* against the environment. In the terminology of game theory [2], 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 $\mathcal{G}$. The achievement design problem can then be understood as asking whether or not there is a *winning strategy* that can be played against the environment *Env* to bring about one of $\mathcal{G}$.

This type 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 [14, pp459–474]. And in fact, we can prove that under certain circumstances, the ACHIEVEMENT AGENT DESIGN decision problem is indeed PSPACE-complete. However, in order to do this, we need to consider how environments are *encoded* or *described* in these decision problems. To understand what is meant by this, consider that the input to our decision problems includes some sort of representation of the behaviour of the environment, and more specifically, the environment's state transformer function τ. Now, one possible description of τ is as a table which maps run/action pairs to the corresponding possible resulting environment states:

$$\begin{array}{lcl} r_1, \alpha_1 & \rightarrow & \{e_1, e_2, \ldots\} \\ \ldots & \rightarrow & \ldots \\ r_n, \alpha_n & \rightarrow & \{\ldots\} \end{array}$$

Such a "verbose" encoding of τ will clearly be exponentially large (in the size of $E \times Ac$), but since the length of runs will be bounded by a polynomial in the size of $E \times Ac$, it will be finite. Once given such an encoding, finding an agent that can be guaranteed to achieve a set of goal states will, however, be comparatively easy. Unfortunately, of course, no such description of the environment will usually be available. In this paper, therefore, we will restrict our attention to environments whose state transformer function is described as a two-tape Turing machine, with the input (a run and an action) written on one tape; the output (the set of possible resultant states) is written on the other tape. It is assumed that to compute the resultant states, the Turing machine requires a number of steps that is at most polynomial in the length of the input[1]. We refer to such environment representations as *concise*.

[1] I am indebted to Paul Dunne for drawing this requirement to my attention, and suggesting the solution.

Given the assumption of concise environment representations, we can prove the following.

Theorem 1 ACHIEVEMENT AGENT DESIGN *is* PSPACE-*complete.*

Proof: The proof involves (i) showing that ACHIEVEMENT AGENT DESIGN is in PSPACE (i.e., there is a polynomial space algorithm that solves the problem), and (ii) showing that a known PSPACE-complete problem can be reduced to ACHIEVEMENT AGENT DESIGN using polynomial time.

For (i), we give the design of a non-deterministic polynomial space Turing machine M that accepts instances of the problem that have a successful outcome, and rejects all others. The inputs to the algorithm will be the task environment $\langle Env, \mathcal{G}\rangle$, together with a run $r = (e_0, \alpha_0, \ldots, \alpha_{k-1}, e_k)$ — the algorithm actually decides whether or not there is an agent that will succeed in the environment given this current run. Initially, the run r will be set to the empty sequence ϵ. The algorithm for M is as follows:

1. if r ends with an environment state in $\mathcal{G}$, then M accepts;
2. if there are no allowable actions given r, then M rejects;
3. non-deterministically choose an action $\alpha \in Ac$, and then for each $e \in \tau(r \cdot \alpha)$ recursively call M with the run $r \cdot \alpha \cdot e$;
4. if all of these accept, then M accepts, otherwise M rejects.

The algorithm thus non-deterministically explores the space of all possible agents, guessing which actions an agent should perform to bring about $\mathcal{G}$. Notice that since any run will be at most polynomial in the size of $E \times Ac$, the depth of recursion stack will be also be at most polynomial in the size of $E \times Ac$. Hence M requires only polynomial space. It follows that ACHIEVEMENT AGENT DESIGN is in NPSPACE (i.e., non-deterministic polynomial space). It only remains to note that PSPACE = NPSPACE [14, p150], and so ACHIEVEMENT AGENT DESIGN is also in PSPACE.

For (ii), we must reduce a known PSPACE-complete problem to ACHIEVEMENT AGENT DESIGN. The problem we choose is that of determining whether a given player has a winning strategy in the game of generalised geography [14, pp460–462]. We refer to this problem as GG. An instance of GG is a triple $\Gamma = \langle N, A, n\rangle$, where N is a set of nodes, $A \subseteq N \times N$ is a directed graph over N, and $n \in N$ is a node in N. GG is a two player game, in which players I and II take it in turns, starting with I, to select an arc (n', n'') in A, where the first arc n' must be the "current node", which at the start of play is n. A move (n', n'') changes the current node to n''. Players are not allowed to visit nodes that have already been visited: play ends when one player, (the loser), has no moves available. The goal of GG is to determine whether player I has a winning strategy.

GG has a similar structure to ACHIEVEMENT AGENT DESIGN, and we can exploit this to produce a simple mapping from instances $\Gamma = \langle N, A, n \rangle$ of GG to the task environments $\langle Env, \mathcal{G} \rangle$ of ACHIEVEMENT AGENT DESIGN. The agent takes the part of player I, the environment takes the part of player II. Begin by setting $E = Ac = N$. We add a further element $e_{\mathcal{G}}$ to E, and define $\mathcal{G}$ to be a singleton containing $e_{\mathcal{G}}$. We now need to define τ, the state transformer function of the environment; the idea is to directly encode the arcs of Γ into τ.

$$\tau(r) = \begin{cases} \emptyset & \text{if } r = (\ldots, n', n'') \text{ and } (n', n'') \notin A \\ \{n\} & \text{if } r = \epsilon \\ \{e_{\mathcal{G}}\} & \text{if } \{n' \mid (last(r), n') \in A \text{ and } n' \notin r\} = \emptyset \\ \{n' \mid (last(r), n') \in A \text{ and } n' \notin r\} & \text{otherwise.} \end{cases}$$

This construction requires a little explanation. The first case deals with the situation where the agent has made an illegal move, in which case the environment disallows any further moves: the game ends without the goal state being achieved. The second case simply defines the environment state corresponding to the first move of GG, i.e., the initial state n of GG. The third case is where player I (represented by the agent) wins, because there are no moves left for player II. In this case the environment returns $e_{\mathcal{G}}$, indicating success. The fourth is the general case, where the environment returns states corresponding to all possible moves. Using this construction, there will exist an agent that can succeed in the environment we construct just in case player I has a winning strategy for the corresponding GG game. Since the construction clearly takes polynomial time, we conclude that ACHIEVEMENT AGENT DESIGN is PSPACE-hard, and we are done. □

The precise relationship of the class PSPACE to the class NP, of problems that may be solved in non-deterministic polynomial time, is not currently known. It *is* known that NP $\subseteq$ PSPACE, and although it is not known whether the inclusion is strict (i.e., NP $\subset$ PSPACE), it is strongly suspected that it *is* strict. It is thus generally believed that PSPACE-complete problems are more complex than NP-complete problems.

Before leaving this section, we can make the following observation about ACHIEVEMENT DESIGN problems: an achievement design problem $\langle Env, \mathcal{G}_1 \rangle$ is *easier* than a problem $\langle Env, \mathcal{G}_2 \rangle$ if $\mathcal{G}_2 \subseteq \mathcal{G}_1$. This leads to the following lemma (the proof is simple).

Lemma 1 *If $\langle Env, \mathcal{G}_1 \rangle$ and $\langle Env, \mathcal{G}_2 \rangle$ are instances of* ACHIEVEMENT AGENT DESIGN *such that $\mathcal{G}_2 \subseteq \mathcal{G}_1$, and there exists some agent that succeeds with respect to $\langle Env, \mathcal{G}_2 \rangle$, then there exists some agent that succeeds with respect to $\langle Env, \mathcal{G}_1 \rangle$.*

3.2 Maintenance Tasks

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 refer to such task environments as *maintenance* task environments.

A task environment with specification Ψ is said to be a maintenance task environment if we can identify some subset $\mathcal{B}$ of environment states, such that $\Psi(r)$ is false if any member of $\mathcal{B}$ occurs in r, and true otherwise. Formally, $\langle Env, \Psi \rangle$ is a maintenance task environment if there is some $\mathcal{B} \subseteq E$ such that $\Psi(r)$ iff for all $e \in \mathcal{B}$, we have $e \notin r$ for all $r \in \mathcal{R}(Ag, Env)$. We refer to $\mathcal{B}$ as the *failure set*. As with achievement task environments, we write $\langle Env, \mathcal{B} \rangle$ to denote a maintenance task environment with environment *Env* and failure set $\mathcal{B}$.

The decision problem for maintenance task environments, corresponding to AGENT DESIGN, is as follows.

> MAINTENANCE AGENT DESIGN
> *Given*: maintenance task environment $\langle Env, \mathcal{B} \rangle$.
> *Answer*: "yes" if there exists an agent Ag that succeeds in $\langle Env, \mathcal{B} \rangle$, "no" otherwise.

It is again useful to think of MAINTENANCE AGENT DESIGN as a game. This time, the agent wins if it manages to *avoid* $\mathcal{B}$. The environment, in the role of opponent, is attempting to force the agent into $\mathcal{B}$; the agent is successful if it has a winning strategy for avoiding $\mathcal{B}$.

Intuition suggests that MAINTENANCE AGENT DESIGN must be *harder* that ACHIEVEMENT AGENT DESIGN. This is because with achievement tasks, the agent is only required to bring about $\mathcal{G}$ *once*, whereas with maintenance tasks environments, the agent must avoid $\mathcal{B}$ *indefinitely*. However, as the following result illustrates, this turns out not to be the case: the problems have the same complexity.

Theorem 2 MAINTENANCE AGENT DESIGN *is* PSPACE-*complete.*

Proof: To show that MAINTENANCE AGENT DESIGN is in PSPACE, we proceed as in Theorem 1 to define a non-deterministic polynomial space Turing machine M that accepts just those instances of the problem that have a "yes" outcome. The construction is similar to Theorem 1. The inputs to M will be the task environment, $\langle Env, \mathcal{B} \rangle$, together with a run $r = (e_0, \alpha_0, \ldots, \alpha_{k-1}, e_k)$. As in Theorem 1, r will initially be the empty sequence ϵ. The algorithm for M is as follows:

1. if r ends with an environment state in $\mathcal{B}$, then M rejects;
2. if there are no allowable actions given r, or if there is an action that ends the run, then M accepts;
3. non-deterministically choose an action $\alpha \in Ac$, then for each $e \in \tau(r \cdot \alpha)$, recursively call M with run $r \cdot \alpha \cdot e$;
4. if all of these accept, then M accepts, otherwise M rejects.

To prove that MAINTENANCE AGENT DESIGN is complete for PSPACE, we again do a reduction from GG. Many details of the reduction are similar to Theorem 1, and so we will omit them. The first point to note is that we create a state $e_{\mathcal{B}}$ and set $\mathcal{B} = \{e_{\mathcal{B}}\}$. In addition, we need to redefine τ:

$$\tau(r) = \begin{cases} \{n\} & \text{if } r = \epsilon \\ \{e_{\mathcal{B}}\} & \text{if } r = (\ldots, n', n'') \text{ and } (n', n'') \notin A \\ \emptyset & \text{if } \{n' \mid (last(r), n') \in A \text{ and } n' \notin r\} = \emptyset \\ \{n' \mid (last(r), n') \in A \text{ and } n' \notin r\} & \text{otherwise.} \end{cases}$$

The first case captures the first move of GG; the second case captures an agent making an illegal move. The third case captures the situation where player II (the environment) has no available moves, in which case the agent wins. The final case is where there are available moves. It is again easy to see that there will be an agent that can avoid $e_{\mathcal{B}}$ just in case the corresponding GG game is a win for player I. Since the construction can be done in polynomial time, it follows that MAINTENANCE AGENT DESIGN is PSPACE-hard, and we are done. □

Notice that a MAINTENANCE AGENT DESIGN problem $\langle Env, \mathcal{B}_1 \rangle$ is easier than a problem $\langle Env, \mathcal{B}_2 \rangle$ if $\mathcal{B}_1 \subseteq \mathcal{B}_2$ (the reverse of the situation for ACHIEVEMENT AGENT DESIGN). The following lemma, corresponding to Lemma 1, is similarly easy to prove.

Lemma 2 *If $\langle Env, \mathcal{B}_1 \rangle$ and $\langle Env, \mathcal{B}_2 \rangle$ are instances of* ACHIEVEMENT AGENT DESIGN *such that $\mathcal{B}_1 \subseteq \mathcal{B}_2$, and there exists some agent that succeeds with respect to $\langle Env, \mathcal{B}_2 \rangle$, then there exists some agent that succeeds with respect to $\langle Env, \mathcal{B}_1 \rangle$.*

4 Automatic Synthesis of Agents

Knowing that there exists an agent which will succeed in a given task environment is helpful, but it would be more helpful if, knowing this, we also had such an agent to hand. How do we obtain such an agent? The obvious answer is to "manually" implement the agent from the specification. However, there are at least two other possibilities (see [19] for a discussion):

1. we can try to develop an algorithm that will *automatically synthesise* such agents for us from task environment specifications; or
2. we can try to develop an algorithm that will *directly execute* agent specifications in order to produce the appropriate behaviour.

In this section, we will briefly consider these possibilities with respect to our framework, focussing primarily on agent synthesis.

Agent synthesis is, in effect, automatic programming: the goal is to have a program that will take as input a task environment, and from this task environment automatically generate an agent that succeeds in this environment. Formally, an agent synthesis algorithm *syn* can be understood as a function

$$syn : \mathcal{TE} \to (\mathcal{AG} \cup \{\perp\}).$$

Note that the function *syn* can output an agent, or else output $\perp$. We will say a synthesis algorithm is *sound* if, whenever it returns an agent, then this agent succeeds in the task environment that is passed as input. We will say *syn* is *complete* if it is guaranteed to return an agent whenever there exists an agent that will succeed in the task environment given as input. Thus a sound and complete synthesis algorithm will only output $\perp$ given input $\langle Env, \Psi \rangle$ when no agent exists that will succeed in $\langle Env, \Psi \rangle$.

Formally, synthesis algorithm *syn* is sound if it satisfies the following condition:

$$syn(\langle Env, \Psi \rangle) = Ag \text{ implies } \mathcal{R}(Ag, Env) = \mathcal{R}_\Psi(Ag, Env).$$

Similarly, *syn* is complete if it satisfies the following condition:

$$\begin{aligned} &\exists Ag \in \mathcal{AG} \text{ s.t. } \mathcal{R}(Ag, Env) = \mathcal{R}_\Psi(Ag, Env) \\ &\quad \text{implies } syn(\langle Env, \Psi \rangle) \neq \perp. \end{aligned}$$

Intuitively, soundness ensures that a synthesis algorithm always delivers agents that do their job correctly, but may not always deliver agents, even where such agents are in principle possible. Completeness ensures that an agent will always be delivered where such an agent is possible, but does not guarantee that these agents will do their job correctly. Ideally, we seek synthesis algorithms that are both sound *and* complete. Of the two conditions, soundness is probably the more important: there is not much point in complete synthesis algorithms that deliver "buggy" agents.

Using the results of this paper, we can make several comments on the computational complexity of agent synthesis algorithms. The first, and most obvious, is that any sound and complete synthesis algorithm implicitly solves a PSPACE-complete problem, since we can use such an algorithm to solve PSPACE-complete agent design problems:

simply give the task environment to the sound and complete synthesis algorithm, and see whether the output is an agent (in which case the answer to the agent design problem is "yes"), or $\perp$ (the answer is "no"). If we are prepared to *relax* either soundness or completeness conditions, then we may be able to obtain an algorithm with more acceptable complexity.

We obtain an interesting perspective on the synthesis of agents if we view task specifications Ψ as formulae of some logical language. In particular, suppose that we have some logic for which models are sequences of states, analogous to our runs. Temporal logic is exactly such a logic [11, 12]: models for (linear, discrete) temporal logic are infinite, linear, discrete sequences of states, similar to our runs (we comment on the use of temporal logic in section 5).

It is easy to see that a specification Ψ will not be implementable if Ψ is unsatisfiable: if Ψ is unsatisfiable, then no run would satisfy Ψ. So, a sound and complete synthesis algorithm can be used as a satisfiability test for a predicate Ψ: if $syn(\langle Env, \Psi \rangle)$ returns an agent, then Ψ is satisfiable. This implies that the computational complexity of *syn* will be at least as bad as the computational complexity of the satisfiability problem for the language in which Ψ is expressed. (In fact — this will perhaps come as no surprise — the satisfiability problem for linear discrete temporal logic, of the kind used in [11, 12], is PSPACE-complete [17].)

If Ψ is expressed in a logical form, then we have the possibility to synthesise agents by doing a *constructive proof of the satisfiability* of specifications. See section 5 for a discussion.

We conclude by noting that an alternative to synthesising agents from specifications is to *directly execute* them. This option has been explored in more detail in the literature. For example, it is the concept that underpins the Concurrent METATEM agent programming language [8]. We leave consideration of direct execution for future work.

5 Related Work

The formal model of agents and environments used in this paper is similar to many that are now used in artificial intelligence, for example, [9, 16].

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 [11, 12]. 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 [15] 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.

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 that realizes the specification [13]. Similar work is reported by Clarke and Emerson [5], who synthesize synchronization skeletons from branching time temporal logic (CTL) specifications.

In artificial intelligence, the planning problem is most closely related to our achievement-based task environments [1]. STRIPS was the archetypal planning system [7]. The STRIPS system is capable of taking a description of the initial environment state e_0, a specification of the goal to be achieved, $\mathcal{G}$, 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 $\mathcal{G}$. 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 $\mathcal{G}$ specified in propositional logic, does there exist a $\pi \in Ac^*$ such that π achieves $\mathcal{G}$?) is PSPACE-complete [4].

More recently, there has been renewed interest by the artificial intelligence planning community in *decision theoretic* approaches to planning [3]. One popular approach involves representing agents and their environments as Partially Observable Markov Decision Processes (POMDPs) [10]. 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, and which is non-deterministic. Finding an optimal policy for a POMDP problem is similar to our agent design problem.

Also closely related is the work of Tennenholtz and Moses on the *multi-entity* model of multi-agent systems [18]. 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

In this paper, we investigated the agent design problem: given a task environment, consisting of an environment together with a task specification, does there exist an agent that will successfully carry out the task in the environment? In particular, we defined two different types of tasks: achievement tasks, where an agent is required to bring about one of a set of goal states, and maintenance tasks, where an agent is required to *avoid* a set of states. We saw that the agent design problem for both types of tasks was PSPACE-complete. In addition, we investigated the implications of this result for the automatic synthesis of agents from task environment specifications.

There are many related problems that demand attention in future work, including for example:

- a precise characterisation of the circumstances under which the agent design problem becomes tractable;
- investigation of the *verification* problem: does agent Ag achieve task Ψ in environment Env?
- investigation of stochastic environments;
- development of efficient synthesis algorithms;
- development of techniques for directly executing agent specifications;
- multi-agent extensions.

Acknowledgements

This paper has benefitted enormously from the suggestions of Paul E. Dunne and Wojciech Rytter.

References

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

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

[3] J. Blythe. An overview of planning under uncertainty. In M. Wooldridge and M. Veloso, editors, *Artificial Intelligence Today (LNAI 1600)*, pages 85–110. Springer-Verlag: Berlin, Germany, 1999.

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

[5] 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.

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

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

[8] 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.

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

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

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

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

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

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

[15] 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, Jan. 1989.

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

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

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

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

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