The Computational Complexity of Agent Verification

Michael Wooldridge and Paul E. Dunne

Department of Computer Science
University of Liverpool
Liverpool L69 7ZF
United Kingdom
{M.J.Wooldridge, P.E.Dunne}@csc.liv.ac.uk

Abstract. In this paper, we investigate the computational complexity of the *agent verification* problem. Informally, this problem can be understood as follows. Given representations of an agent, an environment, and a task we wish the agent to carry out in this environment, can the agent be guaranteed to carry out the task successfully? Following a formal definition of agents, environments, and tasks, we establish two results, which relate the computational complexity of the agent verification problem to the complexity of the task specification (how hard it is to decide whether or not an agent has succeeded). We first show that for tasks with specifications that are in Σ_u^p, the corresponding agent verification problem is Π_{u+1}^p-complete; we then show that for PSPACE-complete task specifications, the corresponding verification problem is no worse — it is also PSPACE-complete. Some variations of these problems are investigated. We then use these results to analyse the computational complexity of various common kinds of tasks, including achievement and maintenance tasks, and tasks that are specified as arbitrary Boolean combinations of achievement and maintenance tasks.

1 Introduction

We share with many other researchers the long-term goal of building agents that will be able to act autonomously in dynamic, unpredictable, uncertain environments in order to carry out complex tasks on our behalf. Central to the deployment of such agents is the problem of *task specification*: we want to be able to describe to an agent what task it is to carry out on our behalf *without* telling the agent *how* it is to do the task.

Associated with the issue of task specification are several fundamental computational problems. The first of these is the *agent design* problem: Given (specifications of) a task and an environment in which this task is to be carried out, is it possible to design an agent that will be guaranteed to successfully accomplish the task in the environment? The complexity of this problem was investigated in [10, 11] for a range of different task types and representation schemes. Two main types of task were investigated in this work: *achievement tasks* and *maintenance tasks*. Crudely, an achievement task is specified by a set of "goal" states;

an agent is simply required to bring about one of these states in order to have succeeded. Maintenance tasks are also specified by a set of states, (referred to as "bad" states), but this time an agent is regarded as having succeeded if it *avoids* these states. The complexity of the agent design problem for both achievement and maintenance tasks was shown to be PSPACE-complete when reasonable assumptions were made about the representation of the task and the environment [10], although less tractable (more verbose) representations reduced the complexity of the problem, rendering it NL-complete in the best case considered. In [11], these results were shown to generalise to seemingly richer types of task specifications: *combined* task specifications, (in which tasks are specified as a pair consisting of an achievement and maintenance task — the agent is required to bring about the goal while maintaining the invariant), and *disjunctive* task specifications, (in which a task is specified as a set of pairs of achievement and corresponding maintenance tasks: on any given run, the agent is required to satisfy one of the goals while maintaining the corresponding invariant).

In this paper, we consider the closely related problem of *agent verification*. Informally, the agent verification problem is as follows. We are presented with (representations of) an environment, a specification of a task to be carried out in this environment, and an agent; we are asked whether or not the agent can be guaranteed to carry out the task successfully in the environment. When considering this problem, we view task specifications as predicates Ψ over execution histories or runs. In this paper, we examine and characterise the complexity of the agent verification problem for two general classes of task specification predicate. In the first case, we examine the complexity of the problem for task specifications Ψ that are (strictly) in Σ_u^p for some $u \in I\!N$. Notice that this includes perhaps the two most important complexity classes: P (i.e., Σ_0^p) and NP (i.e., Σ_1^p). We prove that if Ψ is in Σ_u^p then the corresponding agent verification problem is Π_{u+1}^p-complete. The second general result shows that if the task specification Ψ is PSPACE-complete, then the corresponding verification problem is no worse — it is also PSPACE-complete.

We then use these results to analyse a number of different task specification frameworks. In particular, we analyse the complexity of verification for achievement and maintenance tasks, and the complexity of a more general framework in which tasks are specified as arbitrary Boolean combinations of achievement and maintenance tasks.

We begin in the following section by setting up an abstract model of agents and environments, which we use to formally define the decision problems under study. We then informally motivate and introduce the various agent verification problems we study, and prove our main results. We discuss related work in section 5, and present some conclusions in section 6. Throughout the paper, we presuppose some familiarity with complexity theory [7].

2 Agents, Environments, and Tasks

In this section, we present an abstract formal model of agents, the environments they occupy, and the tasks that they carry out. (Please note that because of space restrictions, we simplified the presentation in this section by omitting some self-evident technical details.)

Environments

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 (over some Ac, E), let $\mathcal{R}^{Ac}$ be the subset of $\mathcal{R}$ that end with an action, and let $\mathcal{R}^E$ be the subset of $\mathcal{R}$ that end with a state. 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:

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

Thus a state transformer function is a partial function that maps a run (assumed to end with an allowable action of the agent) to a set of possible environment states. There are two important points to note about this definition. First, environments are allowed to be *history dependent* (or non-Markovian). 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 previous actions performed by the agent, and the previous environment states 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 the run is *terminated.* Similarly, if an agent has no allowable actions, then we say a run has terminated. One important assumption we make is that every run is guaranteed to terminate with length at most $|Ac \times E|$. This assumption may appear restrictive, and so some justification is necessary. The most important point is that while the restriction is of *theoretical* importance (it limits the generality of our results), it is not likely to be of *practical* importance, for the following reason. Consider an environment whose state is determined by k Boolean values; clearly, this environment can be in 2^k states. In practice, we will *never* have to consider runs that go on for 2^k time steps, for an environment defined by k Boolean attributes. So the restriction on run length, while being a theoretical limit to our work, is reasonable if we are concerned with tasks of practical, everyday interest.

Before proceeding, we need to make clear a couple of assumptions about the way that transformer functions are *represented.* To understand what is meant by this, consider that the input to the decision problems we study will include some sort of representation of the behaviour of the environment, and more specifically, the environment's state transformer function τ. We assume that the state transformer function τ of an environment 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.

Formally, an environment *Env* is a quad $Env = \langle E, Ac, \tau, e_0 \rangle$ where E is a set of environment states, Ac is a set of actions, τ is a state transformer function, and $e_0 \in E$ is the initial state of the environment.

Agents

We model agents simply as functions that map runs (assumed to end with an environment state) to actions (cf. [9, pp580–581]):

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

Notice that while environments are implicitly non-deterministic, agents are assumed to be deterministic.

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*: formally, a sequence $(e_0, \alpha_0, e_1, \alpha_1, e_2, \ldots, e_k)$ represents a run of an agent Ag in environment $Env = \langle E, Ac, \tau, e_0 \rangle$ iff

1. e_0 is the initial state of *Env* and $\alpha_0 = Ag(e_0)$; and
2. for $u > 0$

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

We denote the set of all terminated runs of agent Ag in environment Env by $\mathcal{R}^*(Ag, Env)$, and the set of "least" terminated runs of Ag in Env (i.e., runs r in $\mathcal{R}^*(Ag, Env)$ such that no prefix of r is in $\mathcal{R}^*(Ag, Env)$) by $\mathcal{R}(Ag, Env)$. Hereafter, when we refer to "terminated runs" it should be understood that we mean "least terminated runs".

Tasks

We build agents in order to carry out *tasks* for us. We need to be able to describe the tasks that we want agents to carry out on our behalf; in other words, we need to *specify* them. In this paper, we shall be concerned with tasks that are specified via a predicate Ψ over runs:

$$\Psi : \mathcal{R} \to \{t, f\}.$$

The idea is that $\Psi(r) = t$ means that the run r satisfies specification Ψ, whereas $\Psi(r) = f$ means that run r does not satisfy the specification.

Given a complexity class $\mathcal{C}$, we say that specification Ψ is in $\mathcal{C}$ if the runs that satisfy Ψ form a language that can be recognised in $\mathcal{C}$. In this paper, we will focus primarily on two types of specification: those that are in the class Σ_u^p (for some $u \in I\!N$), and those that are PSPACE-complete. (To be accurate, in the first case we consider task specifications that are *strictly* in Σ_u^p, that is, in $\Sigma_u^p \setminus \Sigma_{u-1}^p$. For convenience, we define $\Sigma_u^p = \emptyset$ for $u < 0$; thus a task specification is in Σ_0^p if it is in $\Sigma_0^p \setminus \emptyset = \mathrm{P} \setminus \emptyset = \mathrm{P}$.) In both cases, we assume that a specification Ψ is represented as a Turing machine T_Ψ that accepts just those runs which satisfy the specification. Any Σ_u^p specification can be represented as an alternating Turing machine that operates in polynomial time using at most u alternations, and similarly, any PSPACE-complete specification can be represented as an polynomial time alternating Turing machine (which must therefore use at most a polynomial number of alternations).

3 Agent Verification

We can now state the agent verification problem:

<u>AGENT VERIFICATION</u>
Given: Environment Env, agent Ag, and task specification Ψ.
Answer: "Yes" if every terminated run of Ag in Env satisfies specification Ψ, "No" otherwise.

This decision problem amounts to determining whether the following second-order formula is true:

$$\forall r \in \mathcal{R}(Ag, Env) \text{ we have } \Psi(r) = t.$$

The complexity of this problem will clearly be determined in part by the complexity of the task specification Ψ. If Ψ is undecidable, for example, then so is the corresponding verification problem. However, for more reasonable task specifications, we can classify the complexity of the verification problem. We focus our attention on the polynomial hierarchy of classes Σ_u^p (where $u \in I\!N$) [7, pp424–425] and the class PSPACE of task specifications that can be decided in polynomial space [7, pp455-489]. (Recall that in the first case, we are concerned with task specifications that are *strictly* in Σ_u^p — see the discussion above.)

Theorem 1. *The agent verification problem for Σ_u^p task specifications is Π_{u+1}^p-complete.*

Proof. To show that the problem is in Π_{u+1}^p, we sketch the design of a Turing machine that decides the problem by making use of a single universal alternation, and invoking a Σ_u^p oracle. The machine takes as input an agent Ag, an environment Env, and a Σ_u^p task specification Ψ:

1. *universally select all runs $r \in \mathcal{R}(Ag, Env)$;*
2. *for each run r, invoke the Σ_u^p oracle to determine the value of $\Psi(r)$;*
3. *accept if $\Psi(r) = t$, reject otherwise.*

Clearly this machine decides the problem in

$$\Pi_1^{p\,\Sigma_u^p} = co\text{-NP}^{\Sigma_u^p} = co\text{-}\Sigma_{u+1}^p = \Pi_{u+1}^p.$$

To show the problem is Π_{u+1}^p-complete, we reduce the problem of determining whether $\text{QBF}_{u+1,\forall}$ formulae are true to agent verification [7, p428]. An instance of $\text{QBF}_{u+1,\forall}$ is given by a quantified Boolean formula with the following structure:

$$\forall \overline{x_1}\ \exists \overline{x_2}\ \forall \overline{x_3}\ \cdots\ Q_{u+1}\overline{x_{u+1}}\ \ \chi(\overline{x_1}, \ldots, \overline{x_{u+1}}) \tag{1}$$

in which:

- *the quantifier Q_{u+1} is "$\forall$" if u is even and "$\exists$" otherwise;*
- *each $\overline{x_i}$ is a finite collection of boolean variables; and*
- *$\chi(\overline{x_1}, \ldots, \overline{x_{u+1}})$ is a propositional logic formula over the Boolean variables $\overline{x_1}, \ldots, \overline{x_{u+1}}$.*

Such a formula is true if for all assignments that we can give to Boolean variables x_1, we can assign values to Boolean variables x_2, ..., such that $\chi(\overline{x_1}, \ldots, \overline{x_{u+1}})$ is true. Here is an example of a $\text{QBF}_{2,\forall}$ formula:

$$\forall x_1 \exists x_2 [(x_1 \vee x_2) \wedge (x_1 \vee \neg x_2)] \tag{2}$$

Formula (2) in fact evaluates to false. (If x_1 is false, there is no value we can give to x_2 that will make the body of the formula true.)

Given a $\text{QBF}_{u+1,\forall}$ formula (1), we produce an instance of agent verification as follows. First, let $\overline{x_1} = x_1^1, x_1^2, \ldots, x_1^m$ be the outermost collection of universally quantified variables. For each of these variables x_1^i, we create two possible environment states, $e_{x_1^i}$ and $e_{\neg x_1^i}$, corresponding to a valuation of true or false respectively to the variable x_1^i. We create an additional environment state e_0, *to serve as the initial state. The environment only allows the agent to perform the action α_0, and the environment responds to the ith performance of action α_0 with either $e_{x_1^i}$ or $e_{\neg x_1^i}$. After m performances of α_0 the run terminates. In this way, each run determines a valuation for the universally quantified variables $\overline{x_1} = x_1^1, x_1^2, \ldots, x_1^m$; the set of all runs corresponds to the set of all possible valuations for these variables. Given a run r, we denote by $\chi(\overline{x_1}, \ldots, \overline{x_{u+1}})[r/\overline{x_1}]$ the Boolean formula obtained from $\chi(\overline{x_1}, \ldots, \overline{x_{u+1}})$ by substituting for each variable x_1^i the valuation (either true or false) made in run r to x_1^i.*

Finally, we define a task specification Ψ as follows:

$$\Psi(r) = \begin{cases} t & \text{if the } \Sigma_u^p \text{ formula } \exists\overline{x_2}\,\forall\overline{x_3}\,\cdots\, Q_{u+1}\overline{x_{u+1}}\,\chi(x_1, \ldots, x_{u+1})[r/\overline{x_1}] \text{ is true} \\ f & \text{otherwise.} \end{cases}$$

Clearly, the input formula (1) will be true if the agent succeeds in satisfying the specification on all runs. Since the reduction is polynomial time, we are done.

We can also consider more expressive task specification predicates: those that are PSPACE-complete. In fact, it turns out that having a PSPACE-complete task specification predicate does not add to the complexity of the overall problem.

Theorem 2. *The agent verification problem for* PSPACE*-complete task specifications is also* PSPACE*-complete.*

Proof. PSPACE*-hardness is obvious from the complexity of the task specification. To show membership of* PSPACE *we first show that the complement of the agent verification problem for* PSPACE*-complete task specifications is in* PSPACE*. The complement of the agent verification problem involves showing that some run does not satisfy the specification, i.e., that there is a run $r \in \mathcal{R}(Ag, Env)$ of agent Ag in environment Env such that $\Psi(r) = f$. The following non-deterministic algorithm decides the problem:*

1. *guess a run $r \in \mathcal{R}(Ag, Env)$, of length at most $|E \times Ac|$;*
2. *verify that $\Psi(r) = f$.*

This algorithm runs in

$$\text{NPSPACE}^{\Pi_1^p} = \text{PSPACE}^{\Pi_1^p} = \text{PSPACE}.$$

*Hence the verification problem is in co-*PSPACE*. But co-*PSPACE=PSPACE *[7, p142], and so we conclude that the agent verification problem for* PSPACE*-complete task specifications is in* PSPACE*.*

An obvious question is whether "simpler" environments can reduce the complexity of the verification problem. An obvious candidate is *deterministic* environments. An environment is deterministic if every element in the range of τ is either a singleton or the empty set. For such environments, the result of performing any action is at most one environment state.

Theorem 3. *Let $\mathcal{C}$ be a complexity class that is closed under polynomial time reductions. Then the verification problem for deterministic environments and task specifications that are in $\mathcal{C}$ is also in $\mathcal{C}$.*

Proof. *Generate the run r of the agent in the environment (which can be done with $O(|E \times Ac|)$ calls on the Turing machine representing the environment, each of which requires at most time polynomial in $|E \times Ac|$), and then verify that $\Psi(r) = t$ (which can be done in $\mathcal{C}$).*

Another obvious simplification is to consider *Markovian* environments: an environment is Markovian, or history independent, if the possible next states of the environment only depend on the current state and the action performed. Although they are intuitively simpler than their non-Markovian counterparts, from the verification point of view, Markovian environments are in fact no better than non-Markovian environments.

Theorem 4. *The agent verification problem for Markovian environments and Σ_u^p strategy specifications is Π_{u+1}^p-complete.*

Proof. *The proof of Theorem 1 will suffice without alteration: the critical point to note is that the environment generated in the reduction from $\text{QBF}_{u+1,\forall}$ is Markovian.*

4 Types of Tasks

In this section, we use the results established in the preceding section to analyse the complexity of verification for three classes of task specifications: achievement and maintenance tasks, and tasks specified as arbitrary Boolean combinations of achievement and maintenance tasks.

4.1 Achievement and Maintenance Tasks

There are two obvious special types of tasks that we need to consider [10]: *achievement tasks* and *maintenance 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. Note that we do not care *which* goal state is achieved — all are considered equally good. Achievement tasks are probably the most commonly studied form of task in artificial intelligence. An achievement task is specified by some subset $\mathcal{G}$ (for "good" or "goal") of environment states E. An agent is *successful* on a particular run 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 $\mathcal{G}$. We say an agent Ag succeeds in an environment Env if every run of the agent in that environment is successful. An agent thus succeeds in an environment if it can guarantee to bring about one of the goal states.

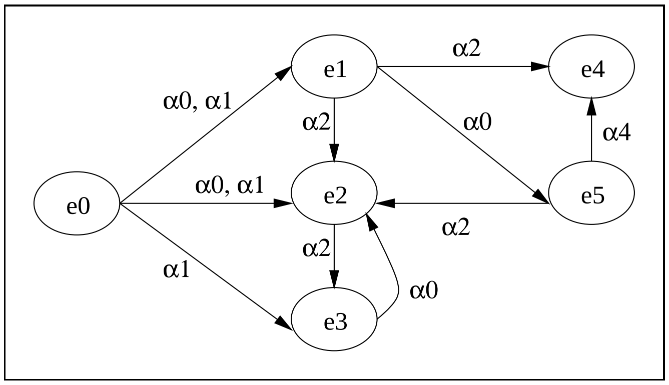

Fig. 1. The state transitions of an example environment: Arcs between environment states are labelled with the sets of actions corresponding to transitions. Note that this environment is *history dependent*, because agents are not allowed to perform the same action twice. So, for example, if the agent reached state e_2 by performing α_0 then α_2, it would not be able to perform α_2 again in order to reach e_3.

Example 1. Consider the environment whose state transformer function is illustrated by the graph in Figure 1. In this environment, an agent has just four available actions (α_1 to α_4 respectively), and the environment can be in any of six states (e_0 to e_5). History dependence in this environment arises because the agent is not allowed to execute the same action twice. Arcs between states in Figure 1 are labelled with the actions that cause the state transitions — note that the environment is non-deterministic. Now consider the achievement tasks determined by the following goal sets:

- $\mathcal{G}_1 = \{e_2\}$
  An agent can reliably achieve $\mathcal{G}_1$ by performing α_1, the result of which will be either e_1, e_2, or e_3. If e_1 results, the agent can perform α_0 to take it to e_5 and then α_2 to take it to e_2. If e_3 results, it can simply perform α_0.
- $\mathcal{G}_2 = \{e_3\}$
  There is no agent that can be guaranteed to achieve $\mathcal{G}_2$. If the agent performs α_1, then any of e_1 to e_3 might result. In particular, if e_1 results, the agent can only get to e_3 by performing α_2 twice, which is not allowed.

Just as many tasks 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, that is, to maintain some invariant condition. 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 tasks as *maintenance* tasks.

A maintenance task is formally defined by a set $\mathcal{B} \subseteq E$ that we refer to as the "bad", or "failure" states — these are the environment states that the agent must avoid. An agent Ag in environment Env is deemed successful with respect to such a maintenance specification if no state in $\mathcal{B}$ occurs on any run in $\mathcal{R}(Ag, Env)$.

Example 2. Consider again the environment in Figure 1, and the maintenance tasks defined by the following bad sets:

– $\mathcal{B}_1 = \{e_5\}$
  There is clearly an agent that can avoid e_5. After the agent performs its first action (either α_0 or α_1), one of the three states e_1 to e_3 will result. If the state that results is e_1, then the agent can perform α_2, after which either e_4 or e_2 will result; there will be no allowable moves in either case. If the state that results is e_2, then the agent can only perform α_2, which will transform the environment to e_4. The only allowable move will then be α_0 (if this has not already been performed — if it has, then there are no allowable moves); if the agent performs α_0, then environment state e_2 will result, from where there will be no allowable moves. Finally, if the state that results is e_3, then the agent can only perform α_0 and then α_2, which returns the environment to state e_3 from where there are no allowable moves.
– $\mathcal{B}_1 = \{e_2\}$
  No agent can be guaranteed to avoid e_2. Whether or not the first action is α_0 or α_1, it is possible that e_2 will result.

Given a an achievement task specification with good set $\mathcal{G}$, or a maintenance task specification with bad set $\mathcal{B}$, it should be clear that the corresponding specifications $\Psi_{\mathcal{G}}$ and $\Psi_{\mathcal{B}}$ will be decidable in (deterministic) polynomial time, i.e., in Σ_0^p. For example, to determine whether a run $r \in \mathcal{R}$ satisfies $\Psi_{\mathcal{G}}$, we simply check whether any member of $\mathcal{G}$ occurs on r. We can therefore apply Theorem 1 to immediately conclude the following.

Corollary 1. *The agent verification problem for achievement and maintenance tasks is Π_1^p-complete (i.e., co-*NP*-complete).*

4.2 Boolean Task Specifications

We can also consider a very natural extension to achievement and maintenance tasks, in which tasks are specified as *arbitrary Boolean combinations* of achievement and maintenance tasks. We specify such tasks via a formula of propositional logic χ. Each primitive proposition p that occurs in χ corresponds to an achievement goal, and hence a set $E_p \subseteq E$ of environment states. A negated proposition $\neg p$ corresponds to a maintenance task with bad states $E_{\neg p} \subseteq E$. From such primitive propositions, we can build up more complex task specifications using the Boolean connectives of propositional logic. To better understand the idea, consider the following example.

Example 3. Recall the environment in Figure 1, and suppose that we allow three primitive proposition letters to be used: p_1, p_2, p_3, where p_1 corresponds to environment states $\{e_2\}$, p_2 corresponds to $\{e_3\}$, and p_3 corresponds to $\{e_1\}$. Consider the following task specifications:

- p_1
  This is an achievement task with goal states $\{e_2\}$. Since p_1 corresponds to $\{e_2\}$, an agent can reliably achieve p_1 by performing α_1, the result of which will be either e_1, e_2, or e_3. If e_1 results, the agent can perform α_0 to take it to e_5 and then α_2 to take it to e_2. If e_3 results, it can simply perform α_0.
- $\neg p_1$
  This is essentially a maintenance task with bad states $\{e_2\}$. Since the agent must perform either α_0 or α_1, no agent can guarantee to avoid e_2.
- $p_1 \vee p_2$
  Since there is an agent that can be guaranteed to succeed with task p_1, there is also an agent that can be guaranteed to succeed with task $p_1 \vee p_2$.
- $p_1 \wedge p_2$
  This task involves achieving *both* e_2 and e_3. There is no agent that can be guaranteed to succeed with this task.
- $p_1 \wedge (p_2 \vee p_3)$
  This task involves achieving e_2 and either e_3 or e_1. There exists an agent that can successfully achieve this task.
- $(p_1 \wedge \neg p_2) \vee (p_2 \wedge \neg p_3)$
  This task involves either achieving e_2 and not e_3 or else achieving e_3 and not e_1. Since there exists an agent that can achieve e_2 and not e_3, there exists an agent that can succeed with this task.

Let us now formalise these ideas. We start from a set $\Phi = \{p_1, \ldots, p_n\}$ of n Boolean variables. A propositional formula $\chi(\Phi_n)$ over Φ_n is inductively defined by the following rules:

1. $\chi(\Phi_n)$ consists of single literal (x or $\neg x$ where $x \in \Phi_n$);
2. $\chi(\Phi_n) = \chi_1(\Phi_n)\ \theta\ \chi_2(\Phi_n)$, where $\theta \in \{\vee, \wedge\}$ and χ_1 and χ_2 are propositional formulae.

Let $f_{\chi(\Phi_n)}$ be the Boolean logic function (of n variables) represented by $\chi(\Phi_n)$. Let $S = \langle E_1, E_2, \ldots, E_n \rangle$ be an ordered collection of *pair-wise disjoint* subsets of E (note that S does *not* have to be a partition of E) and $\chi(\Phi_n)$ be a non-trivial formula. If $r \in \mathcal{R}$ is a run, then the instantiation, $\beta(r) = \langle b_1, b_2, \ldots, b_n \rangle$ of Boolean values to Φ_n *induced by* r is defined by:

$$b_i = \begin{cases} 1 & \text{if } r \text{ contains some state } e \in E_i \\ 0 & \text{otherwise.} \end{cases}$$

In other words, a proposition $p_i \in \Phi_n$ is defined to be true with respect to a run r if one of the states in E_i occurs in r; otherwise, p_i is defined to be false.

A run, r, *succeeds with respect to* the formula χ if $f_\chi(\beta(r)) = 1$. An agent Ag satisfies the Boolean task specification χ in environment Env if $\forall r \in \mathcal{R}(Ag, Env)$

we have $f_\chi(\beta(r)) = 1$. Given a run $r \in \mathcal{R}$ and a Boolean task specification $\chi(\Phi_n)$ the problem of determining whether $f_\chi(\beta(r)) = 1$ is decidable in deterministic polynomial time[1]. So, by Theorem 1, we can conclude:

Corollary 2. *The agent verification problem for Boolean task specifications is Π_1^p-complete.*

5 Related Work

Verification is, of course, a major topic in theoretical computer science and software engineering (see, e.g., [2]). Many techniques have been developed over the past three decades with the goal of enabling efficient formal verification. These approaches can be broadly divided into two categories: *deductive* verification and *model checking*. Deductive approaches trace their origins to the work of Hoare and Floyd on axiomatizing computer programs [4]. Deductively verifying that a system S satisfies some property Ψ, where Ψ is expressed as a formula of some logic $\mathcal{L}$, involves first generating the $\mathcal{L}$-theory $Th(S)$ of S and then using a proof system $\vdash_{\mathcal{L}}$ for $\mathcal{L}$ to establish that $Th(S) \vdash_{\mathcal{L}} \Psi$. The theoretical complexity of the proof problem for any reasonably powerful language $\mathcal{L}$ has been a major barrier to the automation of deductive verification. For example, a language widely proposed for the deductive verification of reactive systems is first-order temporal logic [5, 6]; but proof in first-order temporal logic is not even semi-decidable.

Perhaps closer to our view is the more recent body of work on verification of finite state systems by model checking [3]. The idea is that a system S to be verified can be represented as a model M_S for a branching temporal logic. If we express the specification Ψ as a formula of the temporal logic, then model checking amount to showing that Ψ is valid in M_S: this can be done in deterministic polynomial time for the branching temporal logic CTL [3, p38], wherein lies much of the interest in model checking as a practical approach to verification.

The main difference between our work and that on model checking is that we explicitly allow for a system to be composed of an environment part and an agent part, and our environments are assumed to be much less compliant than is usually the case in model checking (where environments are implicitly both deterministic and Markovian, which allows them to be represented as essentially directed graphs). Some researchers have begun to examine the relationship between model checking and agents in more detail [8, 1], and it would be interesting to examine the relationship of our work to this formally.

6 Conclusions

In this paper, we investigated the complexity of the agent verification problem, that is, the problem of showing that a particular agent, when placed in a particular environment, will satisfy some particular task specification. We established

[1] The proof is by an induction on the structure of $\chi(\Phi_n)$, with the inductive base where $\chi(\Phi_n)$ is a positive literal (an achievement task) or a negative literal (a maintenance task).

two main complexity results for this problem, which allow us to classify the complexity of the problem in terms of the complexity of deciding whether the task specification has been satisfied on any given run of the agent. We first proved that for tasks with specifications that are in Σ_u^p, the corresponding agent verification problem is Π_{u+1}^p-complete; we then showed that for PSPACE-complete task specifications, the corresponding verification problem is also PSPACE-complete. We then used these results to analyse the computational complexity of various common kinds of tasks, including achievement and maintenance tasks and tasks specified as arbitrary Boolean combinations of achievement and maintenance tasks.

There are several important avenues for future work. The first is on the relationship of the verification problem to model checking, as described above. The second is on *probabilistic* verification: Given agent Ag, environment Env, specification Ψ, and probability p, does Ag in Env satisfy Ψ with probability at least p?

Acknowledgments: This research was supported by the EPSRC under grant GR/R60836 ("Algorithmics for agent design and verification").

References

1. M. Benerecetti, F. Giunchiglia, and L. Serafini. Model checking multi-agent systems. *Journal of Logic and Computation*, 8(3):401–424, 1998.
2. R. S. Boyer and J. S. Moore, editors. *The Correctness Problem in Computer Science*. The Academic Press: London, England, 1981.
3. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press: Cambridge, MA, 2000.
4. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, 1969.
5. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag: Berlin, Germany, 1992.
6. Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems — Safety*. Springer-Verlag: Berlin, Germany, 1995.
7. C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley: Reading, MA, 1994.
8. A. S. Rao and M. P. Georgeff. A model-theoretic approach to the verification of situated reasoning systems. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence (IJCAI-93)*, pages 318–324, Chambéry, France, 1993.
9. S. Russell and D. Subramanian. Provably bounded-optimal agents. *Journal of AI Research*, 2:575–609, 1995.
10. M. Wooldridge. The computational complexity of agent design problems. In *Proceedings of the Fourth International Conference on Multi-Agent Systems (ICMAS-2000)*, pages 341–348, Boston, MA, 2000.
11. M. Wooldridge and P. E. Dunne. Optimistic and disjunctive agent design problems. In Y. Lespérance and C. Castelfranchi, editors, *Proceedings of the Seventh International Workshop on Agent Theories, Architectures, and Languages (ATAL-2000)*, 2000.