

APT Agents: Agents That Are Adaptive, Predictable and Timely

Diana F. Gordon

AI Center, Naval Research Laboratory, Washington D.C. 20375

gordon@aic.nrl.navy.mil,

WWW home page: <http://www.aic.nrl.navy.mil/~gordon>

In the Proceedings of the First Goddard Workshop on Formal Approaches to Agent-Based Systems (FAABS'00).

Abstract. The increased prevalence of agents raises numerous practical considerations. This paper addresses three of these – adaptability to unforeseen conditions, behavioral assurance, and timeliness of agent responses. Although these requirements appear contradictory, this paper introduces a paradigm in which all three are simultaneously satisfied. Agent strategies are initially verified. Then they are adapted by learning and formally reverified for behavioral assurance. This paper focuses on improving the time efficiency of reverification after learning. A priori proofs are presented that certain learning operators are guaranteed to preserve important classes of properties. In this case, efficiency is maximal because no reverification is needed. For those learning operators with negative a priori results, we present incremental algorithms that can substantially improve the efficiency of reverification.

1 Introduction

Agents (e.g., robots or softbots) are becoming an increasingly prevalent paradigm. Many systems of the future will be multiagent. The agents paradigm offers numerous advantages, such as flexibility and fault-tolerance. However it also introduces new challenges.

Consider an example. The forthcoming field of nanomedicine holds great promise for microsurgery. Medical nanorobots (also called “nanobot” agents), with tiny sensors and medical devices, will be capable of performing delicate, fine-grained operations within the human body. This would revolutionize the field of medicine. It requires precise navigation through the body, and sophisticated multiagent positioning. For example, multiple nanobots could form a flexible surface comprised of independently controllable agents that translate or rotate their positions relative to each other [8]. Because each human body is unique, these nanobots need to adapt their surgical strategy to the individual.

Unfortunately, by providing agents the capability to adapt, we risk introducing undesirable behavioral side-effects – particularly in situations where global system behavior may be significantly affected by a minor local change. How can

we guarantee that agents will achieve desirable global coordination? For example, we want assurance that the actions of nanobots performing tissue repair will not conflict with the actions of nanobots performing cancer cell removal.

Formal verification can be applied to ensure proper global multiagent coordination. Unfortunately, though, verification can be quite slow. This raises the issue of timeliness. Using the medical nanobots example, agents performing tissue repair could provide formal guarantees that their actions will not conflict with those of other nanobots; yet if the process of verification takes too long, the patient might suffer harm.

In response to problems such as these, we have developed *APT agents*, i.e., agents that are simultaneously adaptive, predictable and timely. Adaptation is achieved with machine learning/evolutionary algorithms, predictability with formal verification, and timeliness by exploiting the knowledge that learning has occurred to streamline the formal verification. To achieve APT agents, we have developed efficient methods for determining whether the behavior of adaptive agents remains within the bounds of pre-specified constraints (called “properties”) after learning. This includes verifying that properties are preserved for single adaptive agents as well as verifying that global properties are preserved for multiagent systems in which one or more agents may adapt.

There has been a growing body of research on learning/adaptive agents (e.g., [14]), as well as evolving agent strategies (e.g., [7]). However, that research does not address formally verifying agents’ behavior following adaptation. Our approach includes both adaptation and formal verification – the latter using model checking. Model checkers can determine whether an agent plan (strategy) S satisfies a property P , i.e., $S \models P$. Although highly effective, model checking has a time complexity problem. Suppose that in a multiagent system, each agent has its own plan. To verify a global property P , the (synchronous) product has to be taken of the individual agent plans to form a multiagent plan S , which is then verified. Model checking global properties of a multiagent plan has time complexity that is exponential in the number of agents. With a large number of agents, this is seriously problematic. In fact, even model checking a single agent plan with a huge number of states can be computationally prohibitive. A great deal of research in the verification community is currently focused on developing reduction techniques for handling very large state spaces [5]. Nevertheless, none of these techniques are tailored specifically for *efficient re-verification* after learning has altered the system (which is the focus of this paper). There are a few methods in the literature that are designed for software that changes. One that emphasizes efficiency, as ours does, is [16]. However none of them (including [16]) are applicable to multiagent systems in which a single agent could adapt, thereby altering the global behavior of the overall system. In contrast, our approach addresses the timeliness of adaptive multiagent systems.

In our APT agents framework (see Figure 1), there are one or more agents with “anytime” plans, i.e., plans that are continually executed in response to internal and external environmental conditions. Each agent’s plan is assumed to

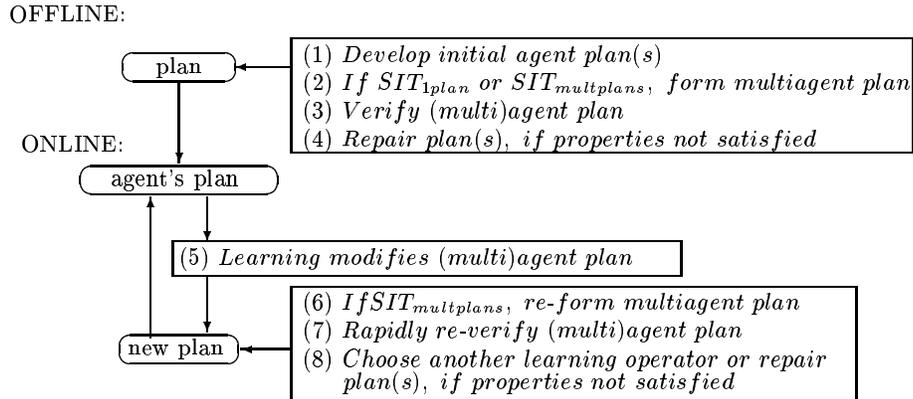


Fig. 1. APT agents framework.

be in the form of a finite-state automaton (FSA). FSAs have been shown to be effective representations of reactive agent plans/strategies (e.g., [4], [7], [11]).

Let us begin with step (1) in Figure 1. There are at least a couple of ways that the FSA plans could be formed initially. For one, a human plan designer could engineer the initial plans. This may require considerable effort and knowledge. An appealing alternative is to evolve (i.e., learn using evolutionary algorithms) the initial plans in a simulated environment [7].

Human plan engineers or evolutionary algorithms can develop plans that satisfy agents' goals to a high degree, but to provide strict behavioral (especially global) guarantees, formal verification is also required. Therefore, we assume that prior to fielding the agents, the (multi)agent plan has been verified offline to determine whether it satisfies critical properties (steps (2) and (3)). These properties can either be expressed in linear temporal logic, or in the form of automata if automata-theoretic (AT) model checking [6] is done. If a property fails to be satisfied, the plan is repaired (step (4)). Steps (2) through (4) require some clarification. If there is a single agent, then it has one FSA plan and that is all that is verified and repaired, if needed. We call this SIT_{1agent} . If there are multiple agents that cooperate, we consider two possibilities. In SIT_{1plan} , every agent uses the same multiagent plan that is the product of the individual agent plans. This multiagent plan is formed and verified to see if it satisfies global multiagent coordination properties. The multiagent plan is repaired if verification produces any errors, i.e., failure of the plan to satisfy a property. In $SIT_{multiplans}$, each agent independently uses its own individual plan. To verify global properties, one of the agents takes the product of these individual plans to form a multiagent plan. This multiagent plan is what is verified. For $SIT_{multiplans}$, one or more individual plans is repaired if the property is not satisfied.

After the initial plan(s) have been verified and repaired, the agents are fielded. While fielded (online), the agents apply learning to their plan(s) as needed (step (5)). Learning (e.g., with evolutionary operators) may be required to adapt the plan to handle unexpected situations or to fine-tune the plan. If SIT_{1agent} or

SIT_{1plan} , the single (multi)agent plan is adapted. If $SIT_{multplans}$, each agent adapts its own FSA, after which the product is formed. For all situations, one agent then rapidly *re-verifies* the new (multi)agent plan to ensure it still satisfies the required properties (steps (6) and (7)). Whenever (re)verification fails, it produces a counterexample that is used to guide the choice of an alternative learning operator or other plan repair as needed (step (8)). This process of executing, adapting, and reverifying plans cycles indefinitely as needed. The main focus of this paper is steps (6) and (7). The novelty of the approach outlined in our framework is not in machine learning or verification per se, but rather the combination of the two.

Rapid reverification after learning is a key to achieving timely agent responses. Our results include proofs that certain useful learning operators are *a priori* guaranteed to be “safe” with respect to important classes of properties, i.e., if the property holds for the plan prior to learning, then it is guaranteed to still hold after learning. If an agent uses these “safe” learning operators, it will be guaranteed to preserve the properties with *no re-verification* required, i.e., steps (6) through (8) in Figure 1 need not be executed. This is the best one could hope for in an online situation where rapid response time is critical. For other learning operators and property classes our a priori results are negative. However, for these cases we have novel *incremental* reverification algorithms that can save time over total reverification from scratch.

2 Agent Plans

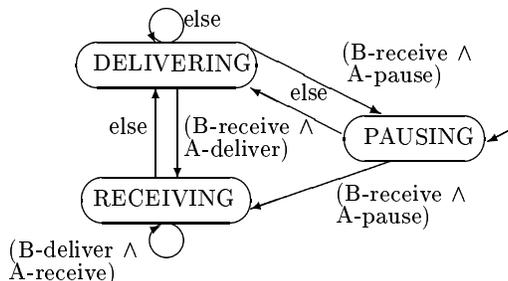


Fig. 2. A finite-state automaton plan for agent A.

Figure 2 shows a finite-state automaton (FSA) plan for an agent that is, perhaps, a nanobot’s protocol for exchanging tissue with another nanobot. The agent may be in a finite number of states, and actions enable it to transition from state to state. Action begins in an initial (i.e., marked by an incoming arrow from nowhere) state (PAUSING in Figure 2). The *transition conditions* (i.e., the Boolean algebra expressions labeling the edges) in an FSA plan succinctly describe the set of actions that enable a state-to-state transition to occur. The operator \wedge means “AND,” \vee means “OR,” and \neg means “NOT.” We use $V(S)$, $I(S)$, $E(S)$, and $M(v_i, v_j)$ to denote the sets of vertices (states), initial

states, edges, and the transition condition from state v_i to state v_j for FSA S , respectively. $\mathcal{L}(S)$ denotes the language of S , i.e., the set of all action sequences that begin in an initial state and satisfy S 's transition conditions. These are the action sequences (called *strings*) allowed by the plan.

Note that the transition conditions of one agent can refer to the actions of one or more other agents. This allows each agent to be reactive to what it has observed other agents doing. Nevertheless, the agents do not have to synchronize on their choice of actions. The only synchronization is the following. At each time step, all of the agents independently choose an action to take. Then they observe the actions of other agents that are referenced in their plan (e.g., agent A's transition conditions mention agent B's actions; therefore A needs to observe what its neighbor, B, did). Based on its own action and those of the other referenced agent(s), the agent knows the next state to which it will transition. The FSAs are assumed to be deterministic and complete, i.e., for every allowable action there is a unique next state.

In SIT_{1plan} or $SIT_{multplans}$, there are multiple agents. Prior to initial verification (and in $SIT_{multplans}$ this is also needed prior to subsequent verification), the synchronous multiagent product FSA is formed. This is done in the standard manner, by taking the Cartesian product of the states and the intersection of the transition conditions. Model checking then consists of verifying that all sequences of multiagent actions allowed by the product plan satisfy the property.

Each multiagent action is an *atom* of the Boolean algebra used in the product FSA transition conditions. To help in understanding the discussions below, we briefly define a Boolean algebra atom. In a Boolean algebra \mathcal{K} , there is a partial order among the elements, \preceq , which is defined as $x \preceq y$ if and only if $x \wedge y = x$. (It may help to think of \preceq as analogous to \subseteq for sets.) The distinguished elements 0 and 1 are defined as $\forall x \in \mathcal{K}, 0 \preceq x$ and $\forall x \in \mathcal{K}, x \preceq 1$. The atoms (analogous to single-element sets) of \mathcal{K} , $\Gamma(\mathcal{K})$, are the nonzero elements of \mathcal{K} minimal with respect to \preceq . For example, (B-receive \wedge A-pause) is an atom, or multiagent action, when there are two agents (A and B). If the agents take multiagent action x , then each agent will transition from its state v_1 to state v_2 whenever $x \preceq M(v_1, v_2)$.

Now we can formalize $\mathcal{L}(S)$. A string (action sequence) \mathbf{x} is an infinite-dimensional vector, $(x_0, \dots) \in \Gamma(\mathcal{K})^\omega$. A *run* \mathbf{v} of string \mathbf{x} is a sequence (v_0, \dots) of vertices such that $\forall i, x_i \preceq M(v_i, v_{i+1})$. In other words, a run of a string is the sequence of vertices visited in an FSA when the string satisfies the transition conditions along the edges. Then $\mathcal{L}(S) = \{\mathbf{x} \in \Gamma(\mathcal{K})^\omega \mid \mathbf{x} \text{ has a run } \mathbf{v} = (v_0, \dots) \text{ in } S \text{ with } v_0 \in I(S)\}$. Such a run is called an *accepting run*, and S is said to *accept* string \mathbf{x} .

3 Adaptive Agents: The Learning Operators

First, let us define the type of machine learning performed by the agents. A machine learning operator $o : S \rightarrow S'$ changes a (product or individual) FSA S to post-learning FSA S' . For a complete taxonomy of our learning operators,

see [9]. Here we do not address learning that adds/deletes/changes states, nor do we address learning that alters the Boolean algebra used in the transition conditions, e.g., via abstraction. Results on abstraction may be found in [10]. Instead, we focus here on edge operators.

Let us begin with our most general learning operator, which we call o_{change} . It is defined as follows. Suppose $z \preceq M(v_1, v_2)$, $z \neq 0$, for $(v_1, v_2) \in E(S)$ and $z \not\preceq M(v_1, v_3)$ for $(v_1, v_3) \in E(S)$. Then $o_{change}(M(v_1, v_2)) = M(v_1, v_2) \wedge \neg z$ (step 1) and/or $o_{change}(M(v_1, v_3)) = M(v_1, v_3) \vee z$ (step 2). In other words, o_{change} may consist of two steps – the first to remove condition z from edge (v_1, v_2) and the second to add condition z to edge (v_1, v_3) . Alternatively, o_{change} may consist of only one of these two steps. Sometimes (e.g., in Subsection 5.3), for simplicity we assume that z is a single atom, in which case o_{change} simply changes the next state after taking action z from v_2 to v_3 . A particular *instance* of this (or any) operator is the result of choosing v_1, v_2, v_3 and z .

Four one-step operators that are special cases of o_{change} are: o_{add} and o_{delete} to add and delete FSA edges (if a transition condition becomes 0, the edge is considered to be deleted), and o_{gen} and o_{spec} to generalize and specialize the transition conditions along an edge. Generalization adds actions to a transition condition (with \vee), whereas specialization removes actions from a transition condition (with \wedge). An example of generalization is the change of the transition condition (B-deliver \wedge A-receive) to ((B-deliver \wedge A-receive) \vee (B-receive \wedge A-receive)). An example of specialization would be changing the transition condition (B-deliver) to (B-deliver \wedge A-receive).

Two-step operators that are special cases of o_{change} are: $o_{delete+gen}$, $o_{spec+gen}$, $o_{delete+add}$, and $o_{spec+add}$. These operators move an edge or part of a transition condition from one outgoing edge of vertex v_1 to another outgoing edge of vertex v_1 . An example, using Figure 2, might be to delete the edge (RECEIVING, RECEIVING) and add its transition condition via generalization to (RECEIVING, DELIVERING). Then the latter edge transition condition would become 1. The two-step operators preserve FSA determinism and completeness. One-step operators must be paired with another operator to preserve these constraints.

When our operators are used in the context of evolving FSAs, each operator application is considered to be a “mutation.” For applicability of the incremental reverification algorithms in Subsection 5.2, we assume that learning is incremental, i.e., at most one operator is applied to one agent per time step (or per generation if using evolutionary algorithms – this is a reasonable mutation rate for these algorithms [1]). Gordon [9] defines how each of the learning operators translates from a single agent FSA to a multiagent product FSA. The only translation we need to be concerned with here is that a single agent o_{gen} may translate to a multiagent o_{add} . We will see the implications of this in Subsections 5.2 and 5.3.

To understand some of the results in Subsection 5.1, it is necessary to first understand the effect that learning operators can have on accessibility. We begin with two basic definitions of accessibility:

Definition 1. Vertex v_n is accessible from vertex v_0 if and only if \exists a path (i.e., a sequence of edges) from v_0 to v_n .

Definition 2. Atom (action) $a_{n-1} \in \Gamma(\mathcal{K})$ is accessible from vertex v_0 if and only if \exists a path from v_0 to v_n and $a_{n-1} \preceq M(v_{n-1}, v_n)$.

Accessibility from initial states is central to model checking, and therefore changes in accessibility introduced by learning should be considered. There are two fundamental ways that our learning operators may affect accessibility: *locally* (abbreviated “L”), i.e., by directly altering the accessibility of atoms or states; *globally* (abbreviated “G”), i.e., by altering the accessibility of any states or atoms that are not part of the learning operator definition. For example, any change in accessibility of v_1, v_2, v_3 , or atoms in $M(v_1, v_2)$ or $M(v_1, v_3)$ in the definition of o_{change} is considered local; other changes are global.

The symbol \uparrow denotes “can increase” accessibility, and \nexists denotes “cannot increase” accessibility. We use these symbols with G and L, e.g., $\uparrow G$ means that a learning operator can (but does not necessarily) increase global accessibility. The following results are useful for Section 5:

- $o_{delete}, o_{spec}, o_{delete+gen}, o_{spec+gen}: \nexists G \nexists L$
- $o_{add}: \uparrow G \uparrow L$
- $o_{gen}: \nexists G \uparrow L$
- $o_{delete+add}, o_{spec+add}, o_{change}: \uparrow G$

Finally, consider a different characterization (partition) of the learning operators, which is necessary for understanding some of the results in Subsection 5.1. For this partition, we distinguish those operators that can introduce at least one new string (action sequence) with an infinitely repeating substring (e.g., (a,b,c,d,e,d,e,d,e,...) where the ellipsis represents infinite repetition of d followed by e) into the FSA language versus those that cannot. The only operators belonging to the second (“cannot”) class are o_{delete} and o_{spec} .

4 Predictable Agents: Formal Verification

Recall that in SIT_{1plan} or $SIT_{multplans}$, there are multiple agents, and prior to initial verification the product FSA is formed (step (2) of Figure 1). In *all three* situations, to perform automata-theoretic (AT) model checking, the product must be taken with the FSA of $\neg P$ for property P (see below). We call the algorithm for forming the product FSA $Total_{prod}$.

Our general verification algorithm (not tailored to learning) is AT model checking. In AT model checking, the negation of the property is expressed as an FSA. Asking whether $S \models P$ is equivalent to asking whether $\mathcal{L}(S) \subseteq \mathcal{L}(P)$ for property P . This is equivalent to $\mathcal{L}(S) \cap \overline{\mathcal{L}(P)} = \emptyset$, which is algorithmically tested by first taking the product (\otimes) of the plan FSA S and the FSA corresponding to $\neg P$, i.e., $S \otimes \neg P$. The FSA corresponding to $\neg P$ accepts $\overline{\mathcal{L}(P)}$. The product

implements language intersection. The algorithm then determines whether $\mathcal{L}(S \otimes \neg P) \neq \emptyset$, which implies $\mathcal{L}(S) \cap \mathcal{L}(P) \neq \emptyset$ ($S \not\models P$). This determination is implemented as a check for undesirable cycles in the product FSA $S \otimes \neg P$ that are accessible from some initial state.

The particular AT verification algorithm we have chosen is a simple, elegant one from Courcoubetis et al. [6]. This algorithm, which we call *Total_{AT}*, is designed to do a depth-first search through the FSA starting at initial states and visiting all states reachable from the initial states – in order to look for all verification errors, i.e., failures to satisfy the property. The algorithm assumes properties are represented as Büchi FSAs [3].¹

We focus mainly on Invariance and Response properties – because these are considered to be particularly relevant for multiagent systems. Invariance properties ($\Box \neg p$, i.e., “always not p ”) can be used to prohibit deleterious multiagent interactions. For example $\Box \neg$ (B-deliver \wedge A-deliver) states that agent B cannot take action B-deliver at the same time that agent A is taking action A-deliver. Response properties ($\Box(p \rightarrow \diamond q)$ i.e., “always if trigger p occurs then eventually response q will occur”) can ensure multiagent coordination by specifying that one agent’s actions follow those of another in a particular sequence. For example \Box (C-deliver $\rightarrow \diamond$ A-receive) states that whenever agent C delivers something A must eventually receive it. Although C is not mentioned in A’s plan, this property can be verified for the three-agent (A, B, C) product FSA.

5 APT Agents: Time-Efficient Reverification

Total reverification is time-consuming. For the sake of timely agents, we first tried to find as many positive a priori results as possible for our operators. Recall that a positive a priori result implies *no* reverification is required.

5.1 A Priori Results

Our objective is to lower the time complexity of reverification. The ideal solution is to identify “*safe*” *machine learning operators* (SMLOs), i.e., machine learning operators that are a priori guaranteed to preserve properties and require no runtime cost. For a plan S and property P , suppose verification has succeeded prior to learning, i.e., $S \models P$. Then a machine learning operator $o(S)$ is an SMLO if and only if verification is guaranteed to succeed after learning, i.e., if $S' = o(S)$, then $S \models P$ implies $S' \models P$.

We next present theoretical results. Proofs for all theorems may be found in [9].² Our two initial theorems, Theorems 1 and 2, which are designed to address the one-step operators, may not be immediately intuitive. For example, it seems reasonable to suspect that if an edge is deleted somewhere along the path from a trigger to a response, then this could cause failure of a Response property to

¹ Because a true Response property cannot be expressed as a Büchi FSA, we use a First-Response property approximation. This suffices for our experiments [9].

² Properties are assumed to be expressed in linear temporal logic or as FSAs.

hold – because the response is no longer accessible. In fact, this is not true. What actually happens is that deletions reduce the number of strings in the language. If the original language satisfies the property then so does is the smaller language. Theorem 1 formalizes this.

Theorem 1. *Let S and S' be FSAs, where S' is identical to S , but with additional edges. We define $o : S \rightarrow S'$ as $o : E(S) \rightarrow E(S')$, where $E(S) \subseteq E(S')$. Then $\mathcal{L}(S) \subseteq \mathcal{L}(S')$.*

Corollary 1. *o_{delete} is an SMLO with respect to any property P .*

Corollary 2. *o_{add} is not necessarily an SMLO for any property, including Invariance and Response properties.*

Theorem 2. *For FSAs S and S' let $o : S \rightarrow S'$ be defined as $o : M(S) \rightarrow M(S')$ where $\exists z \in \mathcal{K}$ (the Boolean algebra), $z \neq 0$, $(v_1, v_2) \in E(S)$, such that $o(M(v_1, v_2)) = M(v_1, v_2) \vee z$. Then $\mathcal{L}(S) \subseteq \mathcal{L}(S')$.*

Corollary 3. *o_{spec} is an SMLO for any property.*

Corollary 4. *o_{gen} is not necessarily an SMLO for any property, including Invariance and Response properties.*

The above theorems and corollaries cover the one-step operators. We next consider theorems that are needed to address the two-step operators. Although we found results for the one-step operators that were general enough to address *any* property, we were unable to do likewise for the two-step operators. Our results for the two-step operators determine whether these operators are necessarily SMLOs for Invariance or Response properties in particular. These results are quite intuitive. The first theorem distinguishes those learning operators that will satisfy Invariance properties from those that will not:

Theorem 3. *A machine learning operator is guaranteed to be an SMLO with respect to any Invariance property P if and only if γG and γL are both true.*

Corollary 5. *Operators $o_{delete+gen}$ and $o_{spec+gen}$ are guaranteed to be SMLOs with respect to any Invariance property.*

Corollary 6. *Operators $o_{delete+add}$, $o_{spec+add}$, o_{change} are not necessarily SMLOs with respect to Invariance properties.*

The next theorem characterizes those learning operators that cannot be guaranteed to be SMLOs with respect to Response properties.

Theorem 4. *Any machine learning operator that can introduce a new string with an infinitely repeating substring into the FSA language cannot be guaranteed to be an SMLO for Response properties.*

Corollary 7. *None of the two-step learning operators is guaranteed to be an SMLO with respect to Response properties.*

For those operators that do not have positive a priori results, we can still save time over total reverification by using incremental algorithms, which are described in the next section.

5.2 Incremental Reverification Algorithms

We just learned that operators o_{spec} and o_{delete} are “safe” learning operators (SMLOs), whereas o_{gen} and o_{add} are not. It is also the case that o_{gen} and o_{add} can cause problems (e.g., for Response properties) when they are part of a two-step operator. Therefore, we have developed incremental reverification algorithms for these two operators.

Recall that there are two ways that operators can alter accessibility: globally (G) or locally (L). Furthermore, recall that o_{add} can increase accessibility either way ($\uparrow G \uparrow L$), whereas o_{gen} can only increase accessibility locally ($\nabla G \uparrow L$). We say that o_{gen} has only a “localized” effect on accessibility, whereas the effects of o_{add} may ripple through many parts of the FSA. The implication is that we can have very efficient incremental methods for reverification tailored for o_{gen} , whereas we cannot do likewise for o_{add} . This is also true for both two-step operators that have o_{gen} as their second step, i.e., $o_{delete+gen}$ and $o_{spec+gen}$. Because no advantage is gained by considering o_{add} per se, we develop incremental reverification algorithms for the most general operator o_{change} . These algorithms apply to all of our operators.

Here we present three incremental algorithms – the first two are for execution after any instance of o_{change} (or its special cases), and the third is only for execution after instances of o_{gen} (or $o_{delete+gen}$ or $o_{spec+gen}$). These algorithms make the assumption that $S \models P$ prior to learning, i.e., any errors found from previous verification have already been fixed. Then learning occurs, i.e., $o(S) = S'$, followed by product re-formation, then incremental reverification (see Figure 1).

The first algorithm is an incremental version of $Total_{prod}$, called Inc_{prod} , which is tailored for re-forming the product FSA (step (6) of Figure 1) after o_{change} has been applied. Recall that in $SIT_{multplans}$ learning is applied to an individual agent FSA, then the product is re-formed. In all situations, the product must be re-formed with the negated property FSA after learning if the type of reverification to be used is AT. Algorithm Inc_{prod} assumes the product was formed originally using $Total_{prod}$. Inc_{prod} capitalizes on the knowledge of which single (or multi)agent state, v_1 , and action, a , have their next state altered by operator o_{change} . (For simplicity, assume a is a multiagent action.) Since the previously generated product is stored, the only product FSA states whose next state needs to be modified are those states that include v_1 and transition on a .

After o_{change} has been applied, followed by Inc_{prod} , incremental model checking is performed. Our incremental model checking algorithm, Inc_{AT} , changes the set of initial states (for the purpose of model checking only) in the product FSA to be the set of all product states formed from state v_1 (whose next state was affected by o_{change}). Reverification begins at these new initial states, rather than the actual initial FSA states. This algorithm also includes another form of streamlining. The only transition taken by the model checker from the new initial states is on action a . This is the transition that was modified by o_{change} . Thereafter, Inc_{AT} proceeds exactly like $Total_{AT}$. Assuming $S \models P$ prior to learning, Inc_{AT} and $Total_{AT}$ will agree on whether $S' \models P$ after learning, whenever P is an Invariance or Response property [9].

We next present an incremental reverification algorithm that is extremely time-efficient. It gains efficiency by being tailored for specific situations (i.e., only in SIT_{1agent} or SIT_{1plan} when there is one FSA to reverify), a specific learning operator (o_{gen}), and a specific class of properties (Response). A similar algorithm for Invariance properties may be found in [10].

The algorithm, called Inc_{gen-R} , is in Figure 3. This algorithm is applicable for operator o_{gen} . However note that it is also applicable for $o_{delete+gen}$ and $o_{spec+gen}$, because according to the a priori results of Subsection 5.1 the first step in these operators is either o_{delete} or o_{spec} which are known to be SMLOs.

Assume the Response property is $P = \Box(p \rightarrow \Diamond q)$ where p is the trigger and q is the response. Suppose property P holds for plan S prior to learning, i.e., $S \models P$. Now we generalize $M(v_1, v_3) = y$ to form S' via $o_{gen}(M(v_1, v_3)) = y \vee z$, where $y \wedge z = 0$ and $y, z \neq 0$. We need to verify that $S' \models P$.

```

procedure check-response-property
if  $y \models q$  then
  if ( $z \models q$  and  $z \models \neg p$ ) then output " $S' \models P$ "
  else output "Avoid this instance of  $o_{gen}$ " fi
else
  if ( $z \models \neg p$ ) then output " $S' \models P$ "
  else output "Avoid this instance of  $o_{gen}$ " fi
fi
end

```

Fig. 3. Inc_{gen-R} reverification algorithm.

The algorithm first checks whether a response could be required of the transition condition $M(v_1, v_3)$. We define a response to be required if for at least one string in $\mathcal{L}(S)$ whose run includes (v_1, v_3) , the prefix of this string before visiting vertex v_1 includes the trigger p not followed by response q , and the string suffix after v_3 does not include the response q either. Such a string satisfies the property if and only if $y \models q$ (i.e., for every atom $a \preceq y$, $a \preceq q$). Thus if $y \models q$ and the property is true prior to learning (i.e., for S), then it is possible that a response is required and thus it must be the case that for the newly added z , $z \models q$ to ensure $S' \models P$. For example, suppose a, b, c, and d are atoms, the transition condition y between STATE4 and STATE5 equals d, and $S \models P$. Let $\mathbf{x} = (a, b, b, d, \dots)$ be an accepting string of S ($\in \mathcal{L}(S)$) that includes STATE4 and STATE5 as the fourth and fifth vertices in its accepting run. The property is $P = \Box(a \rightarrow \Diamond d)$, and therefore $y \models q$ (because $y = q = d$). Suppose o_{gen} generalizes $M(\text{STATE4}, \text{STATE5})$ from d to $(d \vee c)$, where z is c, which adds the string $\mathbf{x}' = (a, b, b, c, \dots)$ to $\mathcal{L}(S')$. Then $z \not\models q$. If the string suffix after (a, b, b, c) does not include d, then there is now a string which includes the trigger but does not include the response, i.e., $S' \not\models P$. Finally, if $y \models q$ and $z \models q$, an extra check is made to be sure $z \models \neg p$ because an atom could be both a response

and a trigger. New triggers are thus avoided. The second part of the algorithm states that if $y \neq q$ and no new triggers are introduced by generalization, then the operator is “safe” to do. It is guaranteed to be safe ($S' \models P$) in this case because a response is not required.

Inc_{gen-R} is a powerful algorithm in terms of its execution speed, but it is based upon the assumption that the learning operator’s effect on accessibility is localized, i.e., that it is o_{gen} with SIT_{1agent} or SIT_{1plan} but not $SIT_{multiplans}$. (Recall that single agent o_{gen} may translate to multiagent o_{add} in the product FSA.) An important advantage of this algorithm is that it never requires forming a product FSA, even with the property. A disadvantage is that it may find false errors. In particular, if $S \models P$ prior to learning and if Inc_{gen-R} concludes that $S' \models P$ after learning, then this conclusion will be correct. However if Inc_{gen-R} finds an error, it may nevertheless be the case that $S' \models P$ [9]. Another disadvantage of Inc_{gen-R} is that it does not allow generalizations that add triggers. If it is desirable to add new triggers during generalization, then one needs to modify Inc_{gen-R} to call Inc_{CAT} when reverification with Inc_{gen-R} fails – instead of outputting “Avoid this instance of o_{gen} .” This modification also fixes the false error problem, *and* preserves the enormous time savings (see next section) when reverification succeeds.

5.3 Empirical Timing Results

Theoretical worst-case time complexity comparisons, as well as the complete set of experiments, are in [9]. Here we present a subset of the results, using Response properties. Before describing the experimental results, let us consider the experimental design.³ The underlying assumption of the design was that these algorithms would be used in the context of evolutionary learning, and therefore the experimental conditions closely mimic those that would be used in this context. FSAs were randomly initialized, subject to a restriction – because the incremental algorithms assume $S \models P$ prior to learning, we restrict the FSAs to comply with this. Another experimental design decision was to show scaleup in the size of the FSAs. Throughout the experiments there were assumed to be three agents, each with the same 12 multiagent actions. Each individual agent FSA had 25 or 45 states. A suite of five Response properties was used (see [9]). The learning operator was o_{change} or o_{gen} . Every algorithm was tested with 30 runs – six independent runs for each of five Response properties. For every one of these runs, a different random seed was used for generating the three FSAs and for generating a new instance of the learning operator. However, it is important to point out that on each run all algorithms being compared with each other used the *same* FSAs, which were modified by the *same* learning operator instance.

Let us consider Tables 1 and 2 of results. Table entries give average cpu times in seconds. Table 1 compares the performance of total reverification with the incremental algorithms that were designed for o_{change} . The situation assumed for these experiments was $SIT_{multiplans}$. Three FSAs were initialized, then the

³ All code was written in C and run on a Sun Ultra 10 workstation.

product was formed. Operator o_{change} , which consisted of a random choice of next state, was then applied to one of the FSAs. Finally, the product FSA was re-formed and re-verification done.

The method for generating Table 2 was similar to that for Table 1, except that o_{gen} was the learning operator and the situation was assumed to be SIT_{1plan} . Operator o_{gen} consisted of a random generalization to the product FSA.

The algorithms (rows) are in triples “p,” “v” and “b” or else as a single item “v=b.” A “p” next to an algorithm name implies it is a product algorithm, a “v” that it is a verification algorithm, and a “b” that it is the sum of the “p” and “v” entries, i.e., the time for *both* re-forming the product and re-verifying. If no product needs to be formed, then the “b” version of the algorithm is identical to the “v” version, in which case there is only one row labeled “v=b.”

Table 1. Average cpu time (in seconds) over 30 runs (5 properties, 6 runs each) with operator o_{change} .

	25-state FSAs	45-state FSAs
Inc_{prod} p	.000574	.001786
$Total_{prod}$ p	.097262	.587496
Inc_{AT} v	.009011	.090824
$Total_{AT}$ v	.024062	.183409
Inc_{AT} b	.009585	.092824
$Total_{AT}$ b	.121324	.770905

Table 2. Average cpu time (in seconds) over 30 runs (5 properties, 6 runs each) with operator o_{gen} .

	25-state FSAs	45-state FSAs
Inc_{prod} p	.000006	.000006
$Total_{prod}$ p	.114825	.704934
Inc_{AT} v	94.660700	2423.550000
$Total_{AT}$ v	96.495400	2870.080000
Inc_{AT} b	94.660706	2423.550006
$Total_{AT}$ b	96.610225	2870.784934
Inc_{gen-R} v=b	.000007	.000006

We tested the hypothesis that the incremental algorithms are faster than the total algorithms – for both product and re-verification. This hypothesis is confirmed in all cases. All differences are statistically significant ($p < .01$, using a Wilcoxon rank-sum test) except those between Inc_{AT} and $Total_{AT}$ in Table 2. In fact, according to a theoretical worst-case complexity analysis [9], in the worst case Inc_{AT} will take as much time as $Total_{AT}$. Nevertheless, in practice it usually provides a reasonable time savings.

What about Inc_{gen-R} , which is even more specifically tailored? First, recall that Inc_{gen-R} can produce false errors. For the results in Table 2, 33% of Inc_{gen-R} 's predictions were wrong (i.e., false errors) for the size 25 FSAs and 50% were wrong for the size 45 FSAs. On the other hand, consider the maximum observable speedup in Tables 1 and 2. By far the best results are with Inc_{gen-R} , which shows a $\frac{1}{2}$ -billion-fold speedup over $Total_{AT}$ on size 45 FSA problems! This alleviates the concern about Inc_{gen-R} 's false error rate – after all, one can afford a 50% false error rate given the speed of trying another learning operator instance and reverifying.

6 Applications

To test our overall framework, we have implemented a simple example of cooperating planetary rovers that have to coordinate their plans. They are modeled as co-evolving agents assuming $SIT_{multipians}$. By using the a priori results and incremental algorithms, we have seen considerable speedups.

We are also developing another implementation that uses reverification during evolution [17]. Two agents compete in a board game, and one of the agents evolves its strategy to improve it. The key lesson that has been learned from this implementation is that although the types of FSAs and learning operators are slightly different from those studied previously, and the property is quite different (it's a check for a certain type of cyclic behavior on the board), initial experiences show that the methodology and basic results here could potentially be easily extended to a variety of multiagent applications.

7 Related and Future Work

This paper has presented efficient methods for behavioral assurance following learning. The incremental reverification algorithms presented here are similar to the idea of local model checking [2] because they localize verification. The difference is that our methods are tailored specifically to learning operators. There is little in the literature about efficient model checking for systems that change. Sokolsky and Smolka [16] is a notable exception – especially since it presents a method for incremental reverification. However, their research is about reverification of software after user edits rather than adaptive multiagent systems.

There is a growing precedent for addressing multiagent coordination by expressing plans as automata and verifying them with model checking (e.g., [13], [4], [11]). Our work builds on this precedent, and also extends it – because none of this previous research addresses efficient *re*-verification for agents that learn.

Finally, there are alternative methods for constraining the behavior of agents, which are complementary to reverification and self-repair. For example, Shoham and Tennenholtz [15] design agents that obey social laws, e.g., safety conventions, by restricting the agents' actions; Spears and Gordon [18] design agents that obey physics laws. Nevertheless, the plan designer may not be able to anticipate and engineer all laws into the agents beforehand, especially if the agents have

to adapt. Therefore, initial engineering of laws should be coupled with efficient reverification after learning.

Future work will focus on extending the a priori results to other learning operators/methods and property classes and other plan representations (such as stochastic/timed FSAs/properties), developing more incremental reverification algorithms, and exploring plan repair to recover from reverification failures [12].

8 Acknowledgements

This research is supported by ONR N0001499WR20010. I am grateful to Bill Spears for numerous useful suggestions.

References

1. Bäck, T. & Schwefel H.-P. (1993). An overview of evolutionary algorithms for parameter optimization. *Evolutionary Computation*, 1(1).
2. Bhat, G. & Cleaveland, R. (1986). Efficient local model checking for fragments of the modal mu-calculus. *Lecture Notes in Computer Science*, 1055.
3. Büchi, J. (1962). On a decision method in restricted second-order arithmetic. *Methodology and Philosophy of Science, Proc. Stanford Int'l Congress*.
4. Burkhard, H. (1993). Liveness and fairness properties in multi-agent systems. *IJCAI'93*.
5. Clarke, E. & Wing, J. (1997). Formal methods: State of the art and future directions. *Computing Surveys*.
6. Courcoubetis, C., Vardi, M., Wolper, M., & Yannakakis, M. (1992). Memory-efficient algorithms for the verification of temporal properties. *Formal Methods in Systems Design*, 1.
7. Fogel, D. (1996). On the relationship between duration of an encounter and the evolution of cooperation in the iterated Prisoner's Dilemma. *Evolutionary Computation*, 3(3).
8. Freitas, Robert, Jr. (1999). Nanomedicine V1: Basic Capabilities. *Landes Bioscience Publishers*.
9. Gordon, D. (2000). Asimovian adaptive agents. *Journal of Artificial Intelligence Research*, 13.
10. Gordon, D. (1998). Well-behaved Borgs, Bolos, and Berserkers. *ICML'98*.
11. Kabanza, F. (1995). Synchronizing multiagent plans using temporal logic specifications. *ICMAS'95*.
12. Kiriakos, K. & Gordon, D. (2000). Adaptive supervisory control of multi-agent systems. *FAABS'00*.
13. Lee, J. & Durfee, E. (1997). On explicit plan languages for coordinating multiagent plan execution. *ATAL'97*.
14. Proceedings of the Workshop on Multiagent Learning (1997). AAAI-97 Workshop.
15. Shoham, Y. & Tennenholtz, M. (1992). Social laws for artificial agent societies: Off-line design. *Artificial Intelligence*, 73.
16. Sokolsky, O. & Smolka, S. (1994). Incremental model checking in the modal mu-calculus. *CAV'94*.
17. Spears, W. & Gordon, D. (2000). Evolving finite-state machine strategies for protecting resources. *ISMIS'00*.
18. Spears, W. & Gordon, D. (1999). Using artificial physics to control agents. *ICHS'99*.