1

Assuring the Behavior of Adaptive Agents

Diana F. Spears

1.1 Introduction

Agents are becoming increasingly prevalent and effective. Robots and softbots, working individually or in concert, can relieve people of a great deal of labor-intensive tedium. Designers can furnish agents with plans to perform desired tasks. Nevertheless, a designer cannot possibly foresee all circumstances that will be encountered by the agent. Therefore, in addition to supplying an agent with a plan, it is essential to also enable the agent to learn and modify its plan to adapt to unforeseen circumstances. The introduction of learning, however, often makes the agent's behavior significantly harder to predict.[1] The goal of this research is to verify the behavior of adaptive agents. In particular, our objective is to develop efficient methods for determining whether the behavior of learning agents remains within the bounds of prespecified 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.

An example of a property is Asimov's First Law [2]. This law, which has also been studied by Weld and Etzioni [34], states that an agent may not harm a human or allow a human to come to harm. The main contribution of Weld and Etzioni is a " 'call to arms:' before we release autonomous agents into real-world environments, we need some credible and computationally tractable means of making them obey Asimov's First Law ... how do we stop our artifacts from causing us harm in the process of obeying our orders?" Of course, this law is too general for direct implementation and needs to be operationalized into specific properties testable on a system, such as "Never delete a user's file." This chapter addresses Weld and Etzioni's call to arms in the context of adaptive agents. To respond to this call, we are developing APT

[1] Even adding a simple, elegant learning mechanism such as chunking in Soar can substantially reduce system predictability (Soar project members, personal communication).

agents, which are agents that are **a**daptive, **p**redictable, and **t**imely. Adaptation is achieved by learning/evolving agent plans, predictability by formal verification, and timeliness by streamlining reverification using the knowledge of what learning was done.

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 agent's 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. 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 developed *incremental* reverification algorithms that can save time over total reverification from scratch.

The novelty of our approach is not in machine learning or verification per se, but rather the synthesis of the two. There are numerous important potential applications of our approach. For example, if antiviruses evolve more effective behaviors to combat viruses, we need to ensure that they do not evolve undesirable virus-like behavior. Another example is data mining agents that can flexibly adapt their plans to dynamic computing environments but whose behavior is adequately constrained for operation within secure or proprietary domains. A third example is planetary rovers that adapt to unforeseen conditions while remaining within critical mission parameters.

The last important application that we will mention is in the domain of power grid and telecommunications networks. The following is an event that occurred (*The New York Times*, September 21, 1991, Business Section). In 1991 in New York, local electric utilities had a demand overload. In attempting to assist in solving the regional shortfall, AT&T put its own generators on the local power grid. This was a manual adaptation, but such adaptations are expected to become increasingly automated in the future. As a result of AT&T's actions, there was a local power overload and AT&T lost its own power, which resulted in a breakdown of the AT&T regional communications network. The regional network breakdown propagated to create a national breakdown in communications systems. This breakdown also triggered failures of many other control networks across the country, such as the air traffic control network. Air travel nationwide was shut down. In the future, it is reasonable to expect that such network controllers will be implemented using multiple, distributed cooperating software agents [15], because global control tends to be too computationally expensive and risky. This example dramatically illustrates the potential vulnerability of our national resources unless these agents satisfy *all* of the following criteria: continuous execution, flexible adaptation to failures, safety, reliability, and timely responses. Our approach ensures that agents satisfy all of these.

1.2 Agent Learning

The ability to adapt is key to survival in dynamic environments. The study of adaptation (learning) in agents is as old as the field of AI. More recently [27], it has blossomed into a field of its own. When agents are adaptive, the following questions need to be addressed:

1. Is learning applied to one or more agents?
2. If multiple agents, are they competing or cooperating?
3. What element(s) of the agent(s) get adapted?
4. What technique(s) are used to adapt?

The answers to questions 1 and 2 are quite variable among published papers. This chapter assumes that learning is applied to one agent at a time, and that the agents cooperate. Regarding question 3, typically it is an agent's plan (strategy) that undergoes adaptation. In other words, learning changes a plan. Example plan representations include action sequences such as in the Procedural Reasoning System (PRS) [10], rule sets [14], finite-state automata [9], or neural networks [21]. Here, we assume finite-state automaton (FSA) plans that map perceptions to actions. Unlike PRS plans, beliefs, goals, and intentions are implicitly embedded in an FSA. FSA plans are stimulus-response, like neural networks or rule-based plans.

Learning alters some component of a plan, for example, it may change the choice of action to take in response to a stimulus. Learning may be motivated by observation, it could be initiated in response to success/failure, or it could be prompted by a general need for performance improvement [13]. The motivation influences the choice of learning technique. Regarding question 4, nearly every learning technique has been used for agents, including analogy, induction, rote, clustering, reinforcement learning (RL), evolutionary algorithms (EAs), backpropagation, and Bayesian updating. The two most prevalent agent learning techniques are RL (e.g., [17, 35]) and EAs (e.g., [1, 24]). A very popular form of RL is Q-learning [33], where agents update their probabilities of taking actions in a given situation based on penalty/reward. This chapter assumes EA learning, as described below. The reason for choosing EA learning is that this is one of the most effective methods known for developing FSA strategies for agents, e.g., see [9]. Unlike RL, EAs can learn the FSA topology.

We assume that learning occurs in two phases: an offline and an online phase. During the offline phase, each agent starts with a randomly initialized population of candidate FSA plans. This population is evolved using the evolutionary algorithm outlined in Figure 1.1. The main loop of this algorithm consists of selecting parent plans from the population, applying perturbation (evolutionary learning) operators to the parents to produce offspring, evaluating the fitness of the offspring, and then returning the offspring to the population if they are sufficiently fit.

```
Procedure EA
t = 0; /* initial generation */
initialize_population(t);
evaluate_fitness(t);
until termination-criterion do
      t = t + 1; /* next generation */
      select_parents(t);
      perturb(t);
      evaluate_fitness(t);
end until
end procedure
```

Fig. 1.1. The outline of an evolutionary algorithm (EA).

At the start of the online phase, each agent selects one "best" (according to its "fitness function") plan from its population for execution. The agents are then fielded and plan execution is interleaved with learning. The purpose of learning during the online phase is to fine-tune the plan and adapt it to keep pace with a shifting environment. The evolutionary algorithm of Figure 1.1 is also used during this phase, but the assumption is a population of size one and incremental learning (i.e., one learning operator applied per plan per generation, which is a reasonable rate for EAs [3]). This is practical for situations in which the environment changes gradually, rather than radically.

1.3 APT Agents Paradigm

This section presents a general APT framework, which assumes cooperating agents. This framework is then illustrated with a detailed example.

1.3.1 General Framework

In our APT agents framework (see Figure 1.2), there are one or more agents with "anytime" plans, i.e., plans that are continually executed in response to internal and external environmental conditions. Let us begin with step (1) in Figure 1.2. There are at least a couple of ways that 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 use machine learning or evolutionary algorithms to develop initial plans.

Human plan engineers or learning 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, such as safety properties (steps (2) and (3)). If a property fails to be satisfied, the plan is repaired (step

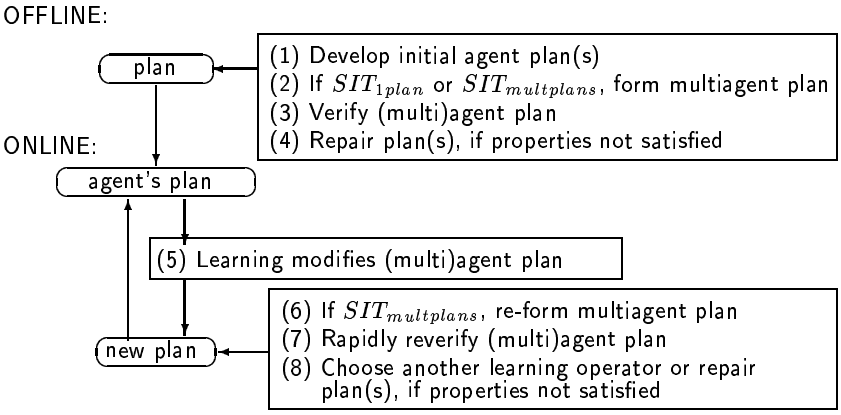

Fig. 1.2. APT agents framework.

(4)). Steps (2) through (4) require some clarification. If there is a single agent, then it has one 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 composition 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 identifies any errors, i.e., failure of the plan to satisfy a property. In $SIT_{multplans}$, each agent independently uses its own individual plan. To verify global properties, one of the agents, which acts as the verification and validation (V&V) agent, takes the composition of these individual plans to form a multiagent plan. This multiagent plan is what is verified. For $SIT_{multplans}$, one or more individual plans are repaired if the property is not satisfied.

After the initial plan(s) have been verified and repaired, the agents are fielded (online) and they apply learning to their plan(s) as needed (step (5)). Learning (e.g., with evolutionary operators) may be required in order 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 plan, after which the composition 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 chapter is steps (6) and (7). Kiriakidis and Gordon [18] address the issue of repairing plans (steps (4) and (8)) when (re)verification errors are found.

We have just presented a general framework. It could be instantiated with a variety of plan representations, learning methods, and verification techniques.

To make this framework concrete, this chapter assumes particular choices for these elements of the framework. In particular, we assume that plans are in the form of finite-state automata, plan learning is accomplished with evolutionary algorithms, the method of verification is model checking [7], and properties are implemented in FSA form, i.e., automata-theoretic (AT) model checking [8] is performed. Model checking consists of building a finite model of a system and checking whether a desired property holds in that model. In the context of this chapter, model checking determines whether $S \models P$ for plan S and property P, i.e., whether plan S "models" (satisfies) property P. The output is either "yes" or "no" and, if "no," one or more counterexamples are provided. Before beginning with the formalisms for these elements, we illustrate them with an example.

1.3.2 Illustrative Example

This subsection presents a multiagent example for SIT_{1plan} and $SIT_{multplans}$ that is used throughout this chapter to illustrate the definitions and ideas. The section starts by addressing $SIT_{multplans}$, where multiple agents have their own independent plans. Later in the section we address SIT_{1plan}, where each agent uses a joint multiagent plan.

Imagine a scenario where a vehicle has landed on a planet for the purpose of exploration and sample collection, for example as in the Pathfinder mission to Mars. Like the Pathfinder, there is a lander (called agent "L") from which a mobile rover emerges. However, in this case there are two rovers: the far ("F") rover for distant exploration, and the intermediary ("I") rover for transferring data and samples from F to L.

We assume an agent designer has developed the initial plans for F, I, and L, shown in Figures 1.3 and 1.4. These are simplified, rather than realistic, plans – for the purpose of illustration. Basically, rover F is either collecting samples/data (in state COLLECTING) or it is delivering them to rover I (when F is in its state DELIVERING). Rover I can either be receiving samples/data from rover F (when I is in its RECEIVING state) or it can deliver them to lander L (when it is in its DELIVERING state). If L is in its RECEIVING state, then it can receive the samples/data from I. Otherwise, L could be busy transmitting data to Earth (in state TRANSMITTING) or pausing between actions (in state PAUSING).

As mentioned above, plans are represented using FSAs. Each FSA has a finite set of states (i.e., the vertices) and allowable state-to-state transitions (i.e., the directed edges between vertices). The purpose of having states is to divide the agent's overall task into subtasks. A state with an incoming arrow not from any other state is an *initial state*. Plan execution begins in an initial state.

Plan execution occurs as the agent takes actions, such as agent F taking action F-collect or F-deliver. Each agent has a repertoire of possible actions, a subset of which may be taken from each of its states. A plan designer can

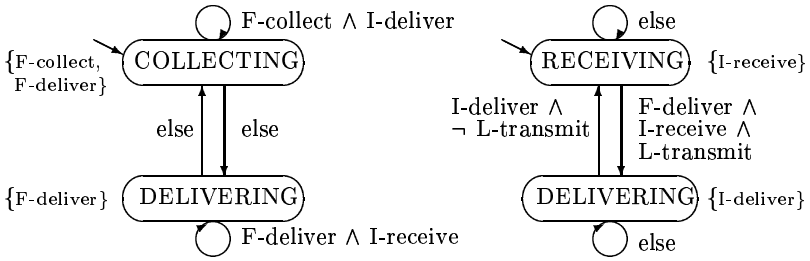

Fig. 1.3. Plans for rovers F (left) and I (right).

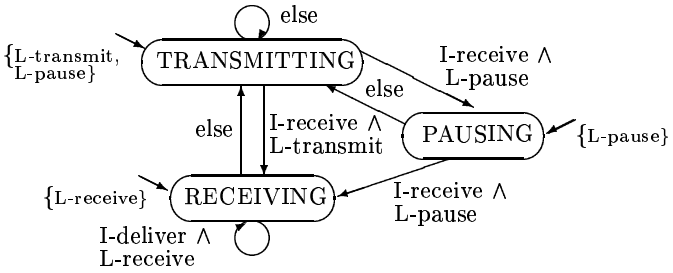

Fig. 1.4. Plan for the lander L.

specify this subset for each state. The choice of a particular action from this subset is modeled as nondeterministic. It is assumed that further criteria, not specified here, are used to make the final run-time choice of a single action from a state.

Let us specify the set of actions for each of the agents (F, I, L) in our example of Figures 1.3 and 1.4. F has two possible actions: F-collect and F-deliver. The first action means that F collects samples and/or data, and the second action means that it delivers these items to I. Rover I also has two actions: I-receive and I-deliver. The first action means I receives samples/data from F, and the second means that it delivers these items to L. L has three actions: L-transmit, L-pause, and L-receive. The first action means L transmits data to Earth, the second that it pauses between operations, and the third that it receives samples/data from I. For each FSA, the set of allowable actions from each state is specified in Figures 1.3 and 1.4 in small font within curly brackets next to the state. For example, rover F can only take action F-deliver from its DELIVERING state.

The *transition conditions* (i.e., the logical expressions labeling the edges) in an FSA plan describe the set of (multiagent) actions that enable a state-to-state transition to occur. The operator ∧ means "AND," ∨ means "OR," and ¬ means "NOT." The condition "else" will be defined shortly. The transition conditions of one agent can refer to the actions of one or more other agents.

This is because each agent is assumed to be reactive to what it has observed other agents doing. If not visible, agents communicate their action choice.

Once an agent's action repertoire and its allowable actions from each state have been defined, "else" can be defined. The transition condition "else" labeling an outgoing edge from a state is an abbreviation denoting the set of all remaining actions that may be taken from the state that are not already covered by other transition conditions. For example, in Figure 1.4, L's three transition conditions from state TRANSMITTING are (I-receive $\wedge$ L-transmit), (I-receive $\wedge$ L-pause), and "else." L can only take the action L-transmit or L-pause from this state. However, rover I could take I-deliver instead of I-receive. Therefore, in this case "else" is equivalent to ((I-deliver $\wedge$ L-transmit) $\vee$ (I-deliver $\wedge$ L-pause)).

Each FSA plan represents a set of allowable action sequences. In particular, a plan is the set of all action sequences that begin in an initial state and obey the transition conditions. An example action sequence allowed by F's plan is $\langle$(F-collect $\wedge$ I-deliver), (F-collect $\wedge$ I-receive), (F-deliver $\wedge$ I-receive), ...$\rangle$ where F takes its actions and observes I's actions at each step in the sequence.

At run-time, these FSA plans are interpreted in the following manner. At every discrete time step, every agent (F, I, L) is at one of the states in its plan, and it selects the next action to take. Agents choose their actions independently. They do not need to synchronize on action choice. The choice of action might be based, for example, on sensory inputs from the environment. Although a complete plan would include the basis for action choice, as mentioned above, here we leave it unspecified in the FSA plans. Our rationale for doing this is that the focus of this chapter is on the verification of properties about correct action sequences. The basis for action choice is irrelevant to these properties.

Once each agent has chosen an action, all agents are assumed to observe the actions of the other agents that are mentioned in its FSA transition conditions. For example, F's transition conditions mention I's actions, so F needs to observe what I did. Based on its own action and those of the other relevant agent(s), an agent knows the next state to which it will transition. There is only one possible next state because the FSAs are assumed to be deterministic. The process of being in a state, choosing an action, observing the actions of other agents, then moving to a next state, is repeated indefinitely.

So far, we have been assuming $SIT_{multplans}$ where each agent has its own individual plan. If we assume SIT_{1plan}, then each agent uses the same multiagent plan to decide its actions. A multiagent plan is formed by taking a "product" (composition) of the plans for F, I, and L. This product models the synchronous behavior of the agents, where "synchronous" means that at each time step every agent takes an action, observes actions of other agents, and then transitions to a next state.[2] Multiagent actions enable state-to-state transitions in the product plan. For example, if the agents jointly take the

[2] The case of an agent not taking an action is represented as action "pause."

actions F-deliver and I-receive and L-transmit, then all agents will transition from the joint state (COLLECTING, RECEIVING, TRANSMITTING) to the joint state (DELIVERING, DELIVERING, RECEIVING) represented by triples of states in the FSAs for F, I, and L. A multiagent plan consists of the set of all action sequences that begin in a joint initial state of the product plan and obey the transition conditions.

Regardless of whether the situation is $SIT_{multplans}$ or SIT_{1plan}, a multiagent plan needs to be formed to verify global multiagent coordination properties (see step 2 of Figure 1.2). Verification of global properties consists of asking whether *all* of the action sequences allowed by the product plan satisfy a property.

One class of (global) properties of particular importance, which is addressed here, is that of forbidden multiagent actions that we want our agents to always avoid, called *Invariance* properties. An example is property P1: $\neg$(I-deliver $\wedge$ L-transmit), which states that it should always be the case that I does not deliver at the same time that L is transmitting. This property prevents problems that may arise from the lander simultaneously receiving new data from I while transmitting older data to Earth. The second important class addressed here is *Response* properties. These properties state that if a particular multiagent action (the "trigger") has occurred, then eventually another multiagent action (the necessary "response") will occur. An example is property P2: If F-deliver has occurred, then eventually L will execute L-receive.

If the plans in Figures 1.3 and 1.4 are combined into a multiagent plan, will this multiagent plan satisfy properties P1 and P2? Answering this question is probably difficult or impossible for most readers if the determination is based on visual inspection of the FSAs. Yet there are only a couple of very small, simple FSAs in this example! This illustrates how even a few simple agents, when interacting, can exhibit complex global behaviors, thereby making global agent behavior difficult to predict. Clearly there is a need for rigorous behavioral guarantees, especially as the number and complexity of agents increases. Model checking fully automates this process. According to our model checker, the product plan for F, I, and L satisfies properties P1 and P2.

Rigorous guarantees are also needed after learning. Suppose lander L's transmitter gets damaged. Then one learning operator that could be applied is to delete L's action L-transmit, which thereafter prevents this action from being taken from state TRANSMITTING. After applying a learning operator, reverification may be required.

In a multiagent situation, what gets modified by learning? Who forms and verifies the product FSA? And who performs repairs if verification fails, and what is repaired? The answers to these questions depend on whether it is SIT_{1plan} or $SIT_{multplans}$. If SIT_{1plan}, the agent with the greatest computational power, e.g., lander L in our example, maintains the product plan by applying learning to it, verifying it, repairing it as needed, and then sending a

copy of it to all of the agents to use. If $SIT_{multplans}$, an agent applies learning to its own individual plan. The individual plans are then sent to the computationally powerful agent, who forms the product and verifies that properties are satisfied. If repairs are needed, one or more agents are instructed to repair their own individual plans.

It is assumed here that machine learning operators are applied one-at-a-time per agent rather than in batch and, if $SIT_{multplans}$, the agents co-evolve plans by taking turns learning [24]. Beyond these assumptions, this chapter does not focus on the learning operators per se (other than to define them). It focuses instead on the outcome resulting from the application of a learning operator. In particular, we address the reverification issue.

1.4 Background

This section gives useful background definitions needed for understanding reverification. We discuss plans, learning, and verification.

1.4.1 Agent Plans

Each agent's plan is assumed to 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., [6, 9, 16]). A particular advantage of the FSA plan representation is that it can be verified with popular verification methods. Example FSAs are shown in Figures 1.3 and 1.4. When using an FSA plan, an agent may be in one of 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. The transition conditions (Boolean algebra expressions labeling FSA edges) in an FSA plan succinctly describe the set of actions that enable a state-to-state transition to occur. 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. 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 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 a 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 $x \wedge y = x$. The distinguished elements 0 and 1 are defined as $\forall x \in \mathcal{K}$, $0 \preceq x \preceq 1$. The atoms of $\mathcal{K}$, $\Gamma(\mathcal{K})$, are the nonzero elements of $\mathcal{K}$ minimal with respect to $\preceq$. For example, suppose there are two agents, A and B. A has two possible actions, A-receive or A-pause, and B has two possible actions, B-receive or B-deliver. Then (A-pause $\wedge$ B-receive) is an atom, or multiagent action. On the other hand (B-receive) is not an atom because it is equivalent to ((A-receive $\wedge$ B-receive) $\vee$ (A-pause $\wedge$ B-receive)). An atom is interpreted as follows. If the agents take multiagent action (atom) 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, $(x_0, ...) \in \Gamma(\mathcal{K})^\omega$. A *run* $\mathbf{v}$ *of string* $\mathbf{x}$ is a sequence $(v_0, ...)$ 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 $\mathcal{L}(S) = \{\mathbf{x} \in \Gamma(\mathcal{K})^\omega \mid \mathbf{x}$ has a run $\mathbf{v} = (v_0, ...)$ in S with $v_0 \in I(S)\}$. Such a run is called an *accepting run*, and S is said to *accept* string $\mathbf{x}$.

1.4.2 Adaptive Agents: The Learning Operators

Adaptation is accomplished by evolving the FSAs (see Figure 1.1). In this subsection we define the evolutionary perturbation operators, also called "machine learning operators." A machine learning operator $o : S \to S'$ changes a (product or individual) FSA S to post-learning FSA S'. For a complete taxonomy of our learning operators, see [11]. Here we do not address learning that changes the set of FSA 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 [12]. 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 the same condition z to edge (v_1, v_3). Alternatively, o_{change} may consist of only one of these two steps. Sometimes 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 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 to change 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 of $o_{delete+gen}$ using the FSA for rover I (Figure 1.3, rightmost FSA) might be to delete the edge (DELIVERING, DELIVERING) and add its transition condition via generalization to (DELIVERING, RECEIVING). Then the latter edge transition condition would become 1. The two-step operators are particularly useful because they preserve FSA determinism and completeness. One-step operators must be paired with another one-step operator to preserve these constraints.

Gordon [11] defines how each of these learning operators translates from a single agent FSA to a multiagent product FSA. This extra translation process is required for $SIT_{multplans}$. The only translation that we need to be concerned with here is that o_{gen} applied to a single FSA may translate to o_{add} in the product FSA. We will see the implications of this in Section 1.6 on incremental reverification.

1.4.3 Two Characterizations of the Learning Operators

To understand some of the a priori theorems about property-preserving operators that are presented in Section 1.5, it is necessary to characterize the learning operators according to the effect that they can have on accessibility. We begin with two basic definitions of accessibility [11]:

Definition 1. *Vertex v_n is accessible from vertex v_0 if and only if there 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 there 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 that are part of the learning operator definition; *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 $\not\uparrow$ 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 characterize the learning operators based on their effect on accessibility:

- o_{delete}, o_{spec}, $o_{delete+gen}$, $o_{spec+gen}$: $\not\downarrow$ G $\not\downarrow$ L
- o_{add}: $\uparrow$ G $\uparrow$ L
- o_{gen}: $\not\downarrow$ G $\uparrow$ L
- $o_{delete+add}$, $o_{spec+add}$, o_{change}: $\uparrow$ G

These results are useful for Sections 1.5 and 1.6, which present theorems and algorithms for efficient reverification.

Finally, consider a different characterization (partition) of the learning operators, which is necessary for understanding some of the results about preservation of Response properties in Section 1.5. 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}.

1.4.4 Predictable Agents: Formal Verification

The verification method assumed here, model checking, consists of building a finite model of a system and checking whether the desired property holds in that model. In other words, model checking determines whether $S \models P$ for plan S and property P, i.e., whether P holds for *every* string (action sequence) in $\mathcal{L}(S)$.

This chapter focuses primarily on Response properties. For results on Invariance properties, and examples of FSA representations of properties, see [11]. Although our implementation uses FSAs for properties, for the sake of succinctness this chapter describes properties with linear temporal logic (LTL) [20]. The general LTL form for Invariance properties is ($\Box\neg p$, i.e., "always not p"). The general LTL form for Response properties is ($\Box(p \rightarrow \Diamond q)$ i.e., "always if trigger p occurs then eventually response q will occur").

1.5 APT Agents: A Priori Results

Total reverification from scratch is time-consuming; it has time complexity exponential in the number of agents. Thus 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 run-time reverification 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 a 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, from [11]. Proofs for all theorems may be found in [11]. 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 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. Therefore, if the original language satisfies the property then so does the smaller language. Theorem 1 formalizes this (in the reverse direction). The corollaries follow immediately from the theorems.

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 a SMLO for any FSA property (which includes Invariance and Response properties).*

Corollary 2. *o_{add} is not necessarily a SMLO for any FSA property.*

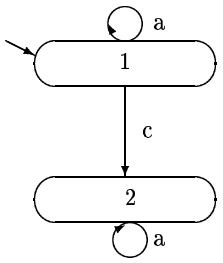

Fig. 1.5. An example to illustrate Corollaries 1 and 2.

Because the intuitions for these corollaries may not be apparent, let us consider an example. Suppose that the following properties must hold for the FSA plan of Figure 1.5:

Invariance: $\Box(\neg b)$

Response: $\Box(c \rightarrow \Diamond a)$

If we delete the lowest edge labeled "a," then both properties still hold. The Response property still holds because a string is an *infinite* sequence of atoms; thus, the only string remaining in the language is an infinite sequence of a's. If we replace the deleted edge by adding a new edge labeled "b," then the Invariance and Response properties are both violated.

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 a SMLO for any FSA property.*

Corollary 4. *o_{gen} is not necessarily a SMLO for any FSA property.*

The intuitions for these corollaries are similar to those for Corollaries 1 and 2.

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* FSA 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. Theorem 3 distinguishes those learning operators that will satisfy Invariance properties from those that will not:

Theorem 3. *A machine learning operator is guaranteed to be a SMLO with respect to any Invariance property P if and only if $\forall\ G$ and $\forall\ L$ 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.*

In other words, increasing local and global accessibility can lead to violation of Invariance properties. Any two-step operator that has "add" as its second step is therefore not necessarily a SMLO.

Theorem 4 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 a SMLO for Response properties.*

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

The problem arises because the infinitely repeating substring can occur between the trigger and the required response.

Operators o_{delete} and o_{spec} are SMLOs, and $o_{delete+gen}$ and $o_{spec+gen}$ are SMLOs for Invariance properties. However, most of these results are negative, where a result is considered to be "negative" if the operator is not a SMLO. In addition to the results here, we have additional results to cover the case where agents learn independently in a multiagent situation ($SIT_{multplans}$)

[11]. However, unfortunately the majority of these results are negative as well. In general, we have found it to be very difficult to find general, positive a priori results. To avoid resorting to the full cost of total reverification, we have explored an alternative solution. In particular, for those operators that do not have positive a priori results, we can still save time over total reverification by using incremental reverification algorithms, which are described in the next section.

1.6 APT Agents: Incremental Reverification

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 ($\not\uparrow$ 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 computational 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.

Prior to reverification, the product FSA must be re-formed (step (2) of Figure 1.2). Subsection 6.2 presents one incremental algorithm for re-forming a product FSA. It also presents two incremental reverification algorithms – the first is for execution after any instance of o_{change}, and the second is only for execution after instances of o_{gen} (or $o_{delete+gen}$ or $o_{spec+gen}$). Both algorithms are sound (i.e., whenever they conclude that reverification succeeds, it is in fact true that $S' \models P$) for Invariance and Response properties. Both are complete for Invariance properties (i.e., whenever they conclude that reverification fails, there is indeed at least one error). However, they are not complete for Response properties, i.e., they may find false errors (property violations). See [11] for proofs regarding soundness and completeness.

Prior to describing the incremental algorithms, we first present the non-incremental (total) versions. These algorithms do not assume that learning has occurred, and they apply to all situations. They are more general (not tailored for learning), but less efficient, than our incremental algorithms.

For implementation efficiency, all of our algorithms assume that FSAs are represented using a table of the transition function $\delta(v_1, a) = v_2$, which means that for state v_1, taking action a leads to next state v_2, as shown in Table 1.1.

Rows correspond to states and columns correspond to multiagent actions. This representation is equivalent to the more visually intuitive representation of Figures 1.3 and 1.4. In particular, Table 1.1 is equivalent to the FSA in Figure 1.4 for the lander agent L. In Table 1.1, states are abbreviated by their first letter, and the multiagent actions are abbreviated by their first letters. For example, "crt" means agent F takes action (F-**c**ollect), I takes (I-**r**eceive), and L takes (L-**t**ransmit). The table consists of entries for the next state, i.e., it corresponds to the transition function. A "0" in the table means that there is no transition for this state-action pair. With this tabular representation, o_{change} is implemented as a perturbation (mutation) operator that changes any table entry to another randomly chosen value for the next state. Operator o_{gen} is a perturbation operator that changes a 0 entry to a next state already appearing in that row. For example, "R" is in row "T" of Table 1.1, so o_{gen} could change "0" to "R" in row "T," column "crr."

Table 1.1. The transition function for agent L's FSA plan. The rows correspond to states and the columns correspond to multiagent actions.

	crt	crr	crp	cdt	cdr	cdp	drt	drr	drp	ddt	ddr	ddp
T	R	0	P	T	0	T	R	0	P	T	0	T
R	0	T	0	0	R	0	0	T	0	0	R	0
P	0	0	R	0	0	T	0	0	R	0	0	T

1.6.1 Total Algorithms

The algorithms described in this subsection make no assumption about learning having occurred, i.e., they are general-purpose verification methods. The particular automata theoretic (AT) verification algorithm we have chosen is from Courcoubetis et al. [8]. This algorithm, called $Total_{AT}$, does depth-first search through the FSA starting at initial states and visiting all reachable 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 [5]. [3]

In AT model checking, 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 an 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 $\mathcal{L}(S) \cap \overline{\mathcal{L}(P)} \neq \emptyset$ ($S \not\models P$).

[3] Because the negation of a Response property cannot be expressed as a Büchi FSA, we use a First-Response property approximation. This suffices for our experiments [11].

Suppose there are n agents, and $1 \leq j_k \leq$ the number of states in the FSA for agent k. Then the algorithm forms all product states $v = (v_{j_1}, .., v_{j_n})$ and specifies their transitions:

```
Procedure product
for each product state v = (v_{j_1}, .., v_{j_n}) do
    if all v_{j_k}, 1 ≤ k ≤ n, are initial states, then v is a product initial state
    endif
    for each multiagent action a_i do
        if (δ(v_{j_k}, a_i) == 0) for some k, 1 ≤ k ≤ n, then δ(v, a_i) = 0
        else δ(v, a_i) = (δ(v_{j_1}, a_i),..., δ(v_{j_n}, a_i)); endif
    endfor
endfor
end procedure
```

Fig. 1.6. $Total_{prod}$ product algorithm.

To implement AT verification, we first need to form the product FSA $S \otimes \neg P$, which equals $S_1 \otimes ... \otimes S_n$, where $S_1, ..., S_{n-1}$ are the agents' plans and S_n equals $\neg P$. The algorithm for doing this, $Total_{prod}$ shown in Figure 1.6, forms product states and then specifies the transition function for these states.

Once the product FSA has been formed, it can be verified. Figure 1.7 shows the algorithm from Courcoubetis et al. [8] that we use. We call this algorithm $Total_{AT}$ because it is total automata-theoretic verification. Algorithm $Total_{AT}$ checks whether $S \not\models P$ for any property P, i.e., whether $\mathcal{L}(S \otimes \neg P) \neq \emptyset$. This is true if there is some "bad" state in a special set of states $B(S \otimes \neg P)$ that is reachable from an initial state and reachable from itself, i.e., part of an accessible cycle and therefore visited infinitely often. The algorithm of Figure 1.7 performs this check using a nested depth-first search on the product $S \otimes \neg P$. The first depth-first search begins at initial states and visits all accessible states. Whenever a state $s \in B(S \otimes \neg P)$ is discovered, it is called a "seed," and a nested search begins to look for a cycle that returns to the seed. If there is a cycle, this implies the $B(S \otimes \neg P)$ (seed) state can be visited infinitely often, and therefore the language is nonempty (i.e., there is some action sequence in the plan that does not satisfy the property) and verification fails.

1.6.2 Incremental Algorithms

Algorithm $Total_{AT}$ can be streamlined if it is known that learning occurred. The incremental reverification 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 reformation, then incremental reverification (see Figure 1.2).

```
Procedure verify
    for each state v ∈ V(S ⊗ ¬P) do
        visited(v) = 0
    endfor
    for each initial state v ∈ I(S ⊗ ¬P) do
        if (visited(v) == 0) then dfs(v); endif
    endfor
end procedure
Procedure dfs(v)
    visited(v) = 1;
    if v ∈ B(S ⊗ ¬P) then
        seed = v;
        for each state v ∈ V(S ⊗ ¬P) do
            visited2(v) = 0
        endfor
        ndfs(v)
    endif
    for each successor (i.e., next state) w of v do
        if (visited(w) == 0) then dfs(w); endif
    endfor
end procedure
Procedure ndfs(v) /* the nested search */
    visited2(v) = 1;
    for each successor (i.e., next state) w of v do
        if (w == seed) then print "Bad cycle. Verification error";
        break
        else if (visited2(w) == 0) then ndfs(w); endif
        endif
    endfor
end procedure
```

Fig. 1.7. $Total_{AT}$ verification algorithm.

The first algorithm is an incremental version of $Total_{prod}$, called Inc_{prod}, shown in Figure 1.8, which is tailored for re-forming the product FSA (step (6) of Figure 1.2) after o_{change}. For simplicity, all of our algorithms assume o_{change} is applied to a single atom, which we assume is a multiagent action. Since we use the tabular representation, this translates to changing one table entry. 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 (multi)agent state, v_1, and multiagent action, a, have their next state altered by operator o_{change}. Since the previously generated product is stored, the only product

```
Procedure product
I(S) = ∅;
for each product state v = (v_{j1}, ..., v_i, ..., v_{jn}) formed from state v_i do
    if visited(v) then I(S) = I(S) ∪ {v}; endif
    if (δ(v_{jk}, a_adapt) == 0) for some k, 1 ≤ k ≤ n, then δ(v, a_adapt) = 0
    else δ(v, a_adapt) = (δ(v_{j1}, a_adapt),..., w_i',..., δ(v_{jn}, a_adapt)); endif
endfor
end procedure
```

Fig. 1.8. Inc_{prod} product algorithm.

```
Procedure verify
    for each state v ∈ V(S ⊗ ¬P) do
        visited(v) = 0
    endfor
    for each new initial state v ∈ I(S ⊗ ¬P) do
        if (visited(v) == 0) then dfs(v); endif
    endfor
end procedure
Procedure dfs(v)
    visited(v) = 1;
    if v ∈ B(S ⊗ ¬P) then
        seed = v;
        for each state v ∈ V(S ⊗ ¬P) do
            visited2(v) = 0
        endfor
        ndfs(v)
    endif
    if v ∈ I(S ⊗ ¬P) and w ≠ 0 and (visited(w) == 0),
    where w = δ(v, a_adapt), then dfs(w)
    else
        for each successor (i.e., next state) w of v do
            if (visited(w) == 0) then dfs(w); endif
        endfor
    endif
end procedure
```

Fig. 1.9. Procedures verify and dfs of the Inc_{AT} reverification algorithm. Procedure ndfs is the same as in Figure 1.7.

FSA states whose next state is 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} shown in Figure 1.9, changes the set of initial states (*only* during model checking) 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 $Total_{AT}$.

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 [12].

```
Procedure check-response-property
    if y ⊨ q then
        if (z ⊨ q and z ⊨ ¬p) then output "S' ⊨ P"
        else output "Avoid this instance of o_gen"; endif
    else
        if (z ⊨ ¬p) then output "S' ⊨ P"
        else output "Avoid this instance of o_gen"; endif
    endif
end procedure
```

Fig. 1.10. Inc_{gen-R} reverification algorithm.

The algorithm, called Inc_{gen-R}, is in Figure 1.10. 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 Section 1.5 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 $S' \models P$.

The algorithm first checks whether a response could be required of the new 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 $S \models P$, where $P = \Box$ (a $\rightarrow \Diamond$ d). Let $\mathbf{x}$ = (a, b, b, d, ...) 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. $P = \Box$ (a $\rightarrow \Diamond$ d), and therefore

$y \models q$ (because $y = q =$ d). Suppose o_{gen} generalizes M(STATE4, STATE5) from d to (d $\vee$ c), where z is c, which adds the string $\mathbf{x}' =$ (a, b, b, c, ...) 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 $y \not\models q$ 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 $SIT_{multplans}$. (Recall from Subsection 4.2 on the learning operators 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. 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_{AT} 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.

Finally, we also have two incremental algorithms, Inc_I and Inc_{gen-I}, described in Gordon [11]. These algorithms do model checking that is not automata-theoretic, but is instead streamlined specifically for Invariance properties.

1.6.3 Time Complexity Evaluation

Theoretical worst-case time complexity comparisons, as well as the complete set of experiments, are in [11]. Here we present a subset of the results, using Response properties. Before describing the experimental results, let us consider the experimental design. [4] 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. In keeping with this design assumption, 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.[5] Throughout the experiments there were assumed to be three agents,

[4] All code was written in C and run on a Sun Ultra 10 workstation.

[5] Our first objective was to see how the reverification performs on FSAs large enough to handle real-world applications. Future experiments will focus on scaling up the number of agents.

with each agent's plan using the same 12 multiagent actions. Each individual agent FSA had 25 or 45 states. A suite of five Response properties was used (see [11]). The learning operator was o_{change} or o_{gen}. Although the two-step $o_{delete+gen}$ and $o_{spec+gen}$ are more useful than o_{gen}, only the generalization step can generate verification errors. Thus we use the simpler o_{gen} in the experiments. 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 1.2 and 1.3 of results. Table entries give average (arithmetic mean) cpu times in seconds. Table 1.2 compares the performance of total reverification with the incremental algorithms that were designed for o_{change}. The situation assumed for these experiments was $SIT_{multplans}$. Three FSAs were initialized, then the 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 reverification done.

The method for generating Table 1.3 was similar to that for Table 1.2, 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" implies that the entry is the sum of the "p" and "v" entries, i.e., the time for *both* re-forming the product and reverifying. 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 1.2. Average cpu time (in seconds) over 30 runs (5 Response 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

Before we present and address the experimental hypothesis, let us first address why Table 1.3 has higher cpu times for Inc_{AT} and $Total_{AT}$ than Table 1.2. This disparity is caused by a difference in the number of verification errors (property violations). In particular, no errors occurred after o_{change}, but

Table 1.3. Average cpu time (in seconds) over 30 runs (5 Response 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

errors did occur after o_{gen}. The lack or presence of errors in these experiments is a by-product of the particular random FSA that happened to be generated, as well as the choice of learning operator. In the future, we plan to analyze *why* certain operators tend to generate more errors than others for particular classes of properties.

Now let us present and address the experimental hypothesis. We tested the hypothesis that the *incremental* algorithms are faster than the *total* algorithms – for both product and reverification. As shown in the tables, 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 1.3. In fact, according to a theoretical worst-case complexity analysis [11], in the worst case Inc_{AT} will take as much time as $Total_{AT}$. Nevertheless we have found in practice, for our applications, that 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.[6] For the results in Table 1.3, 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 1.2 and 1.3. 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.

It is possible to infer the performance of the incremental algorithms for the two-step operators based on their performance with the one-step operators. For example, we know that o_{delete} is a SMLO. We also know that the most efficient reverification algorithm following o_{gen} for a Response property is Inc_{gen-R}. Combining these two results, we can conclude that Inc_{gen-R} will also be the most efficient reverification algorithm to use for $o_{delete+gen}$. We conclude this section by summarizing, in Table 1.4, the fastest algorithm (based on our results) for every two-step operator (since they tend to be

[6] False errors are not a problem for Inc_{AT} in these experiments [11].

Table 1.4. Learning operators with the fastest reverification method.

	$SIT_{1agent/1plan}$ and Invariance	$SIT_{1agent/1plan}$ and Response	$SIT_{multplans}$ and Invariance	$SIT_{multplans}$ and Response
o_{change}	Inc_I	Inc_{AT}	Inc_I	Inc_{AT}
$o_{delete+add}$	Inc_I	Inc_{AT}	Inc_I	Inc_{AT}
$o_{spec+add}$	Inc_I	Inc_{AT}	Inc_I	Inc_{AT}
$o_{delete+gen}$	*None*	Inc_{gen-R}	Inc_I	Inc_{AT}
$o_{spec+gen}$	*None*	Inc_{gen-R}	Inc_I	Inc_{AT}

more useful than one-step operators), situation, and property type. Results for Inc_I are discussed in [11]. In Table 1.4, "None" means no reverification is required, i.e., the learning operator is a priori guaranteed to be a SMLO for this situation and property class.

1.7 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 $SIT_{multplans}$. By using the a priori results and incremental algorithms, we have seen considerable speedup.

We are also developing another implementation that uses reverification during evolution [30]. Two agents compete in a board game, and one of the agents evolves its plan 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 experiments show that the methodology and basic results here could potentially be easily extended to a variety of multiagent applications.

1.8 Related Research

The FAABS'00 proceedings [26] provides an excellent sample of current research on predictable agents – including examples of significant applications, e.g., see Pecheur and Simmons [23]. For related work on predictable agents whose plans are expressed as finite-state automata and verified with model checking, Lee and Durfee [19] is a good example. Nevertheless, almost none of this research addresses adaptive agents.

In fact, even the issue of verifying adaptive *software* has been largely neglected. The research of Sokolsky and Smolka [29] 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.

Three papers in the FAABS'00 proceedings consider verifiable adaptive agents. The first, by Zhu [36], assumes that all possible adaptations are known a priori and can be captured in the agent's (verifiable) plan. The second, by Owre et al. [22], considers applicability of the Symbolic Analysis Laboratory (SAL) for re-verifying agents' plans after adaptations such as determinization have altered the plans. Efficiency is considered, thus making this an APT agents architecture. The third paper, by Kiriakidis and Gordon [18], describes an APT agents architecture somewhat different from the one in this chapter. To alleviate much of the overhead incurred when repairing plans if reverification fails, Kiriakidis and Gordon formulate their framework within the paradigm of discrete supervisory control [25].

Barley and Guesgen [4] address an interesting issue pertaining to APT agents. They determine whether agents' adaptations preserve completeness (coverage), In particular, they specify conditions under which it is guaranteed that solutions found by an agent's problem solver will remain solutions – in spite of adaptation.

Finally, there are alternative methods for constraining the behavior of agents, which are complementary to reverification and self-repair. Turney [32] mentions a wide variety of possible alternatives. "Laws" can constrain behavior, e.g., Shoham and Tennenholtz [28] design agents that obey *social* laws, and Spears and Gordon [31] design agents that obey *physics* laws. Although laws constrain behavior, a plan designer may not be able to anticipate all needed laws beforehand – especially if the agents have to adapt. Therefore, initial engineering of laws should be coupled with efficient reverification after learning.

1.9 Summary and Open Problems

To handle real-world domains and interactions with people, agents must be adaptive, predictable, *and* rapidly responsive. An approach to resolving these potentially conflicting requirements is presented here. In summary, we have shown that certain machine learning operators are a priori (with no run-time reverification) safe to perform, i.e., they are property preserving. All of the a priori results are independent of the size of the FSA and are therefore applicable to any FSA that has been model checked originally.

We then presented novel incremental reverification algorithms for the cases in which the a priori results are negative. Experimental results are shown which indicate that these algorithms can substantially improve the time complexity of reverification over total reverification from scratch. One of the algorithms showed as much as a $\frac{1}{2}$-billion-fold speedup on average over traditional verification.

The significance of this work is that any early offline assurances about agent behavior risk being rendered obsolete by subsequent adaptation. What

has been shown here is that for some adaptations reverification can be localized to just the region of the plan affected by the change, thus making retesting potentially very fast. Furthermore, we have identified property-preserving adaptations that require no reverification, i.e., behavioral assurance is guaranteed to be preserved without the need to retest it. A limitation of this research is that not all properties, plans, and learning methods are amenable to localization. Our results are applicable only when such localization is possible.

A great deal remains to be done on APT agents. A recent RIACS/NASA Ames Workshop on the Verification and Validation of Autonomous and Adaptive Systems (Asilomar, CA; December 2000) raised intriguing questions that could lead to fruitful future research (http://ase.arc.nasa.gov/vv2000/asilomar-report.html).

One important direction for future research would be to explore how to achieve APT agents in the context of a variety of different agent architectures or classes of relevant agent properties/constraints. For example, this chapter assumes that agent plans are represented as deterministic finite-state automata. However there are many other options, such as stochastic finite-state automata or rule sets. The Automated Software Engineering Group at NASA Ames is currently pursuing APT-related research in the context of neural network architectures. Regarding classes of properties, Manna and Pneuli [20] provide a taxonomy of temporal logic properties that could be considered.

Also, recall that one of the agents in a multiagent situation acts as the V&V agent. Is it possible to distribute the V&V burden among multiple agents? We plan to investigate this question.

Another profitable direction for future research is to explore the wide variety of machine learning methods potentially used by APT agents [27]. The list of possible learning techniques is broad and includes statistical learning, case-based inference, reinforcement learning, Bayesian updating, evolutionary learning, inductive inference, speedup learning, and theory refinement. These learning methods enable adaptation to changing environmental conditions, and they can also increase the efficiency and/or effectiveness of plans used by agents.

Finally, deeper analyses of *why* certain learning techniques are better/worse (in terms of the number of verification errors they cause) for particular classes of properties could be quite beneficial. Such analyses could inspire new a priori results, new incremental reverification algorithms, as well as new "safe" learning algorithms.

Acknowledgements

This research was supported by the ONR contract N0001499WR20010 and performed while I was employed at the Naval Research Laboratory. I am grateful to the anonymous reviewers for their helpful comments, and I am

especially grateful to Bill Spears for numerous constructive suggestions. The presentation of this material was improved enormously thanks to Bill's advice.

The figures and tables in this chapter are reprinted from my earlier article entitled "Asimovian adaptive agents," in the *Journal of Artificial Intelligence Research, Volume 13*, copyright 2000, pages 95-153, with permission from the publisher Elsevier.

References

1. Arora, N. and Sen, S. Resolving social dilemmas using genetic algorithms. In *Proceedings of the AAAI Symposium on Adaptation, Coevolution and Learning in Multiagent Systems*, AAAI Press, Stanford, CA, 1996, pp 1–5.
2. Asimov, I. *I, Robot.* Fawcett Publications, Inc., Greenwich, CT. 1950.
3. Bäck, T. and Schwefel, H. P. An overview of evolutionary algorithms for parameter optimization. *Evolutionary Computation*, **1**(1):1–24, MIT Press, 1993.
4. Barley, M. and Guesgen, H. *Towards safe learning agents.* Technical Report, University of Auckland, 2001.
5. Büchi, J. On a decision method in restricted second-order arithmetic. In *Methodology and Philosophy of Science, Proceedings of the Stanford International Congress*, Stanford University Press, Stanford, CA, 1962, pp 1–11.
6. Burkhard, H. Liveness and fairness properties in multi-agent systems. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence (IJCAI'93)*, Morgan-Kaufmann, Chambéry, France, 1993, pp 325–330.
7. Clarke, E. and Wing, J. Formal methods: State of the art and future directions. *ACM Computing Surveys*, **28**(4):626–643, Association for Computing Machinery Publishers, 1997.
8. Courcoubetis, C., Vardi, M., Wolper, M. and Yannakakis, M. Memory-efficient algorithms for the verification of temporal properties. *Formal Methods in Systems Design*, **1**:257–288, Kluwer, 1992.
9. Fogel, D. On the relationship between duration of an encounter and the evolution of cooperation in the iterated Prisoner's Dilemma. *Evolutionary Computation*, **3**(3):349–363, MIT Press, 1996.
10. Georgeoff, M. and Lansky, A. Reactive reasoning and planning. In *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI'87)*, Seattle, WA, Morgan-Kaufmann, 1987, pp 677–682.
11. Gordon, D. Asimovian adaptive agents. *Journal of Artificial Intelligence Research*, **13**:95–153, Morgan-Kauffman, 2000.
12. Gordon, D. Well-behaved Borgs, Bolos, and Berserkers. In *Proceedings of the 15th International Conference on Machine Learning (ICML'98)*, Madison, WI, Morgan-Kaufmann, 1998, pp 224–232.
13. Grecu, D. and Brown, D. Dimensions of learning in agent-based design. In *Proceedings of the 4th International Conference on AI in Design – Workshop on "Machine Learning in Design"*, Stanford, CA, AAAI Press, 1996, pp 21–26.
14. Grefenstette, J., Ramsey, C. and Schultz, A. Learning sequential decision rules using simulation models and competition. *Machine Learning*, **5**(4):355–382, Kluwer, 1990.

15. Jennings, R., Mamdani, E., Corera, J., Laresgoiti, I., Perriolat, F., Skarek, P. and Varga, L. Using ARCHON to develop real-world DAI applications. *IEEE Expert*, **11**(6):64–70, IEEE Computer Society Publishers, 1996.
16. Kabanza, F. Synchronizing multiagent plans using temporal logic specifications. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS'95)*, San Francisco, CA, MIT Press, 1995, pp 217–224.
17. Kaelbling, L, Littman, M. and Moore, A. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, **4**:237–285, Morgan-Kauffman, 1996.
18. Kiriakidis, K. and Gordon, D. Adaptive supervisory control of multi-agent systems. In *Formal Approaches to Agent-Based Systems (FAABS'00), (LNAI 1871)*, Greenbelt, MD, Springer-Verlag, 2001, pp 304–305.
19. Lee, J. and Durfee, E. On explicit plan languages for coordinating multiagent plan execution. In *Proceedings of the 4th Workshop on Agent Theories, Architectures, and Languages (ATAL'97), (LNAI 1365)*, Providence, RI, Springer-Verlag, 1997, pp 113–126.
20. Manna, Z. and Pnueli, A. The anchored version of the temporal framework. *Lecture Notes in Computer Science*, **345**:201–284, Springer-Verlag, 1989.
21. Miller, W., Sutton, R. and Werbos, P. (editors). *Neural Networks for Control.* MIT Press, Cambridge, MA. 1992.
22. Owre, S., Ruess, H., Rushby, J. and Shankar, N. Formal approaches to agent-based systems with SAL. In *Formal Approaches to Agent-Based Systems (FAABS'00) Abstracts*, Greenbelt, MD, 2001.
23. Pecheur, C. and Simmons, R. From Livingstone to SMV: Formal verification for autonomous spacecrafts. In *Formal Approaches to Agent-Based Systems (FAABS'00), (LNAI 1871)*, Greenbelt, MD, Springer-Verlag, 2001, pp 103–113.
24. Potter, M., Meeden, L. and Schultz, A. Heterogeneity in the coevolved behaviors of mobile robots: The emergence of specialists. In *Proceedings of the 17th International Conference on Artificial Intelligence (IJCAI'01)*, Seattle, WA, Morgan-Kaufmann, 2001, pp 1337–1343.
25. Ramadge, P. and Wonham, W. Supervisory control of a class of discrete event processes. *SIAM Journal of Control and Optimization*, **25**(1):206–230, Globe Publishers, 1987.
26. Rash, J, Rouff, C., Truszkowski, W., Gordon D. and Hinchey, M. (editors). In *Formal Approaches to Agent-Based Systems (FAABS'00), (LNAI 1871)*, Greenbelt, MD, Springer-Verlag, 2001.
27. Sen, S. (editor). In *Proceedings of the AAAI Symposium on Adaptation, Coevolution and Learning in Multiagent Systems*, AAAI Spring Symposium, Stanford, CA, AAAI Press, 1996.
28. Shoham, Y. and Tennenholtz, M. Social laws for artificial agent societies: Offline design. *Artificial Intelligence*, **73**(1-2):231–252, Elsevier Science Publishers, 1995.
29. Sokolsky, O. and Smolka, S. Incremental model checking in the modal mu-calculus. In *Proceedings of the Sixth Conference on Computer-Aided Verification (CAV'94), (LNCS 818)*, Stanford, CA, Springer-Verlag, 1994, pp 351–363.
30. Spears, W. and Gordon, D. Evolving finite-state machine strategies for protecting resources. In *Proceedings of the Foundations of Intelligent Systems (ISMIS'00), (LNAI 1932)*, Charlotte, NC, Springer-Verlag, 2000, pp 166–175.
31. Spears, W. and Gordon, D. Using artificial physics to control agents. In *Proceedings of the IEEE International Conference on Information, Intelligence and*

Systems (ICIIS'99), Bethesda, MD, IEEE Computer Society Publishers, 1999, pp 281–288.

32. Turney, P. Controlling super-intelligent machines. *Canadian Artificial Intelligence*, **27**:3–35, Canadian Society for Computational Studies of Intelligence Publishers, 1991.
33. Watkins, C. *Learning from delayed rewards.* Doctoral dissertation, Cambridge University, Cambridge, England, 1989.
34. Weld, D. and Etzioni, O. The first law of robotics. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, Seattle, WA, MIT Press, 1994, pp 1042–1047.
35. Wolpert, D., Wheeler, K. and Tumer, K. General principles of learning-based multi-agent systems. In *Proceedings of the Third International Conference on Autonomous Agents*, Seattle, WA, Association for Computing Machinery Publishers, 1999, pp 77–83.
36. Zhu, H. Formal specification of agent behaviors through environment scenarios. In *Formal Approaches to Agent-Based Systems (FAABS'00), (LNAI 1871)*, Greenbelt, MD, Springer-Verlag, 2001, pp 263–277.