*In the Proceedings of the 15th International Conference on Machine Learning (ICML-98)*

# Well-Behaved Borgs, Bolos, and Berserkers

**Diana F. Gordon**
Naval Research Laboratory, Code 5510
4555 Overlook Avenue, S.W.
Washington, DC 20375
gordon@aic.nrl.navy.mil

## Abstract

How can we *guarantee* that our software and robotic agents will behave as we require, even after learning? Formal verification should play a key role but can be computationally expensive, particularly if *re*-verification follows each instance of learning. This is especially a problem if the agents need to make rapid decisions and learn quickly while online. Therefore, this paper presents novel methods for reducing the time complexity of re-verification subsequent to learning. The goal is agents that are predictable *and* can respond quickly to new situations.

## 1 INTRODUCTION

Software and robotic agents are becoming increasingly prevalent. Agent designers can furnish such 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 plans, it is essential to also enable the agent to learn and modify its plans to adapt to unforeseen circumstances. The introduction of learning, on the other hand, often makes the agent's behavior significantly harder to predict. Our objective is to develop methods that provide verifiable guarantees that the behavior of learning agents always remains within the bounds of specified constraints (called "properties"), even after learning. An example of a property is Asimov's First Law of Robotics (Asimov, 1942). This law, which has recently been studied by Weld and Etzioni (1994), states that a robot may not harm a human or allow a human to come to harm. Weld and Etzioni advocate 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?" Asimov's law can be operationalized into specific properties testable on a system, e.g., "Never delete another user's file." This paper addresses Weld and Etzioni's "call to arms" in the context of adaptive agents. It is a very important topic for real-world agents and is a dominant theme in science fiction, which is sometimes prescient. Examples include the Borgs (Star Trek, The New Generation), Bolos (Laumer, 1976), and Berserkers (Saberhagen, 1967) – fictional agents that demonstrate the dangerous behavior that can result from insufficient constraints.

We assume that an agent's plan has been initially verified offline. Then, the agent is fielded and has to adapt online. After adaptation via learning, the agent must rapidly *re*-verify its new plan to ensure this plan still satisfies required properties.[1] Re-verification must be as computationally efficient as possible because it is performed online, perhaps in a highly time-critical situation. There are numerous applications of this scenario, including software agents that can safely access information in confidential or proprietary environments while responding to rapidly changing access requirements, planetary rovers that quickly adapt to unforeseen planetary conditions but behave within critical mission constraints, and JAVA applets that can get smarter but not become destructive to our computing environments.

Typically, properties desired by a user are orthogonal to the agent's planning goals and to its learning goals.

[1]Current output is success/failure. Future work will consider using re-verification counterexamples to choose a better learning method when re-verification fails.

For example, the agent may generate a plan with the objective of maximizing the agent's profit. Learning might have the goal of achieving the agent's plan more efficiently or modifying the plan to adapt to unforeseen events. The designer also may have a constraint that the agent does not cheat in its dealings with other agents. Why doesn't the planner incorporate all properties into the plan? There are a number of possible reasons, e.g., not all properties may be known at the time the plan is developed, or security reasons.

Re-verification can be (from least to most time required): none, incremental, or complete. It is possible to avoid re-verification entirely if we restrict the agent to using only those learning methods determined a priori to be "safe" with respect to certain classes of properties in which we are interested. In other words, if a plan satisfies a property prior to learning, we want an a priori guarantee that the property will still be satisfied subsequent to learning. Note that this incurs *no* run-time cost. It is called "moving a tester into the generator" or "compiling constraints."

Unfortunately, the safety of some learning methods may be very difficult or maybe impossible to determine a priori. When a priori determination is too difficult, it is helpful to use incremental re-verification. Incremental methods save computational costs over re-verification from scratch by localizing re-verification and/or by reusing knowledge from the original verification. Furthermore, incremental methods may identify positive results that cannot be determined a priori. When an agent needs to learn, we suggest that the agent should consult the a priori results first. If no positive results exist, then incremental re-verification proceeds. The least desirable of the three alternatives is to do complete re-verification from scratch.

Gordon (1997a) begins to explore the extent to which we can prove a priori results that certain machine learning operators are, or are not, safe for certain classes of properties. The paper has positive a priori results for plan efficiency improvements via deletion of plan elements, as well as for plan refinement methods. Unfortunately, we have not yet obtained positive a priori results for popular machine learning operators such as abstraction (unless one is willing to accept an abstracted property) or generalization. Abstraction is a more global operator than generalization. Abstraction alters the language of a plan (e.g., by feature selection), whereas generalization alters the condition for a state-to-state transition within a plan. Both are extremely common operators in concept learning, but are also very appropriate for plan modification.

This paper has two contributions beyond (Gordon, 1997a). First, the previous paper models agent plans using automata on infinite strings. This paper reaches a wider audience by using the more familiar automata on finite strings. Second, this paper addresses two, new questions: Are there situations in which an abstracted property is acceptable? If yes, we have positive a priori results for abstraction. Also, can we get positive results by using incremental re-verification rather than a priori? Initial, positive answers to these questions are presented here.

The remainder of this paper is organized as follows. Section 2 presents an illustrative example that is used throughout the paper. [2] Section 3 contains background material and definitions on automaton plans, temporal logic properties, and "safe" learning. The formal definitions provide a precise foundation for understanding the incremental re-verification methods presented later. Section 4 lists situations in which property abstraction is acceptable. Sections 4 and 5 present novel (and as far as we are aware, the only) methods for incremental re-verification of abstraction and generalization, respectively, on automata. Finally, time complexity comparisons between incremental and complete re-verification are provided.

## 2 ILLUSTRATIVE EXAMPLE

This section provides an example to illustrate some of the main ideas of the paper. Although the plan in this example is very small, it is important to point out that existing automata-based verification methods currently handle huge, industrial-sized problems (e.g., see Kurshan, 1994). Our goal is to improve the time complexity of verification over current methods when learning occurs.

In our example, hundreds of tiny, micro air vehicles (MAVs) are required to perform a task within a region. The MAVs are divided into two groups called "swarm A" and "swarm B." One constraint, or property, is that only one MAV may enter the region at a time – because multiple MAVs entering simultaneously would increase the risk of detection. Each swarm has a separate FIFO queue of MAVs. MAVs enter the queue when they return from their last task. A second constraint is that some (at least one) MAVs from each swarm eventually enter the region. One distinguished MAV, C, acts as a

[2]Examples in this paper have been implemented using Kurshan's COSPAN verification system. COSPAN is an AT&T verification tool, which is described in Kurshan (1994).

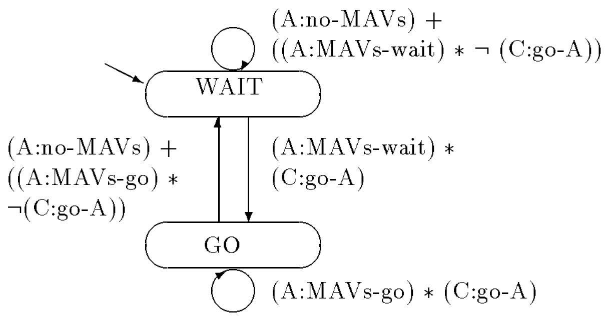

Figure 1: Plan A

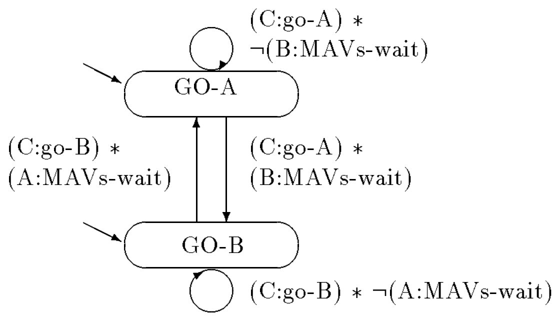

Figure 2: Plan C

task coordinator. C selects which swarm, A or B, may send in an MAV next.[3]

Plans for swarm A and task controller C are shown in Figures 1 and 2. The plan for swarm B is not shown in the figure, but it is identical to the plan for A except all instances of "A" are replaced by "B." Each of these plans is a finite-state automaton, i.e., a graph with states (the vertices) and allowable state-to-state transitions (the directed edges between vertices). The transition conditions (i.e., the logical expressions labeling the edges) describe the set of actions that enable a state transition to occur. The possible actions A can take from a state are (A:no-MAVs), (A:MAVs-wait), or (A:MAVs-go). The first action means the queue is empty, the second that the queue is not empty but the MAVs in the queue must wait, and the third that the first MAV in the queue enters the region. Likewise for B. The possible actions C can take from a state are (C:go-A) or (C:go-B). The first action means controller C allows swarm A to send one MAV into the region, the second means C allows B to send one MAV into the region.

Swarms A and B are single agents, i.e., although individual MAVs may each have their own plan, such as queuing within a swarm, for simplicity we ignore that level of detail. We can form a multiagent plan by taking a "product" (see Section 3.1) of the plans for A, B, and C. This product synchronizes the behavior of A, B, and C in a coordinated fashion. At every discrete time step, every agent (A, B, C) is at one state in its plan, and it selects its next action. The action of one agent (e.g., A) becomes an input to the other agents' plans (e.g., B and C). If the joint actions chosen by all three agents satisfy the transition conditions of a plan from the current state to some next state, then that transition may be made. For example, if the agents jointly take the actions (A:MAVs-wait) and (B:MAVs-wait) and (C:go-A), then the multiagent plan can transition from the global, joint state (WAIT, WAIT, GO-A) to the joint state (GO, WAIT, GO-B) represented by triples of states in the automata for agents A, B, and C.

Given the full, multiagent plan, verification now consists of asking the question: Does this plan satisfy the two required properties, i.e., some MAVs from each swarm enter the region, but only one MAV enters the region at a time? Assuming our initial plan in Figures 1 and 2 satisfies these properties, we next ask whether the properties are still satisfied subsequent to learning. The latter question is the topic of this paper.

An example of learning is the following. Suppose coordinator C discovers that the B swarm has left the region. One way agent C can adapt to incorporate this new knowledge is by deleting the action (C:go-B) from its action repertoire. This is a form of abstraction. There are alternative modifications agent C can do, but the selection between these alternatives is a learning issue, which we do not address here. What we do address here are the implications of this choice, in particular, which learning methods are safe, i.e., preserve the properties.

## 3 PLANS, PROPERTIES, AND "SAFE" LEARNING

### 3.1 AUTOMATON PLANS

This subsection, which is based on Kurshan (1994), briefly summarizes the basics of the automata used to model plans. Figures 1 and 2 illustrate the definitions. Essentially, an automaton is a graph with vertices corresponding to states and directed edges corresponding to state-to-state transitions. The terms "vertex" and "state" are used interchangeably throughout

[3]This example is a variant of the traffic controller in Kurshan (1994).

the paper. For an automaton representing an agent's plan, vertices represent the internal state of the agent and/or the state of its external environment. State-to-state transitions have associated transition conditions, which are the conditions under which the transition may be made. An agent action that satisfies a transition condition enables that transition to be made. We assume finite-state automata, i.e., the set of states is finite, and that the transition conditions are elements of a Boolean algebra. Therefore, we briefly digress to summarize the basics of Boolean algebras.

A Boolean algebra $\mathcal{K}$ is a set with distinguished elements 0 and 1, closed under the Boolean operations $*$ (logical "and"), $+$ (logical "or"), and $\neg$ (logical negation), and satisfying the standard properties (Kurshan, 1994).

The Boolean algebras are assumed to be finite. There is a partial order among the elements, $\preceq$, which is defined as $x \preceq y$ if and only if $x * y = x$. The 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* of $\mathcal{K}$, $\Gamma(\mathcal{K})$, are the nonzero elements of $\mathcal{K}$ minimal with respect to $\preceq$. For two different atoms $x$ and $y$ within the *same* Boolean algebra, $x * y = 0$. For Figures 1 and 2, agents A, B, and C each have their own Boolean algebra with its atoms. The atoms of A's Boolean algebra are the actions (A:no-MAVs), (A:MAVs-wait), and (A:MAVs-go); the atoms of B's algebra are (B:no-MAVs), (B:MAVs-wait), and (B:MAVs-go); the atoms of C's algebra are (C:go-A) and (C:go-B).

A Boolean algebra $\mathcal{K}'$ is a *subalgebra* of $\mathcal{K}$ if $\mathcal{K}'$ is a non-empty subset of $\mathcal{K}$ that is closed under the operations $*$, $+$, and $\neg$, and also has the distinguished elements 0, 1. Let $\mathcal{K} = \prod \mathcal{K}_i$, i.e., $\mathcal{K}$ is the product algebra of the $\mathcal{K}_i$. In this case the $\mathcal{K}_i$ are subalgebras of $\mathcal{K}$. An atom of the product algebra is the product of the atoms of the subalgebras. For example, if $a_1, ..., a_n$ are atoms of subalgebras $\mathcal{K}_1, ..., \mathcal{K}_n$, respectively, then $a_1 * ... * a_n$ is an atom of $\mathcal{K}$.

In Figure 1, the Boolean algebra $\mathcal{A}$ used by agent A is the smallest one containing the atoms of A's algebra. It contains all Boolean elements formed from A's atoms using the Boolean operators $*$, $+$, and $\neg$, including 0 and 1. These same definitions hold for B and C's algebras $\mathcal{B}$ and $\mathcal{C}$. One atom of the product algebra $\mathcal{ABC}$ is (A:no-MAVs) $*$ (B:no-MAVs) $*$ (C:go-A). This is the form of actions taken by the three agents in the multiagent plan. Algebras $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$ are subalgebras of the product algebra $\mathcal{ABC}$. Finally, $\mathcal{ABC}$ is the Boolean algebra for the transition conditions in the multiagent plan.

Let us return now to automata. This paper focuses on automata that model agents with finite lifetimes (represented as a finite string, or sequence of actions). An example is an agent that is created specially to execute a plan and is destroyed immediately afterwards. In particular, we focus on *processes*. Processes are automata, but they are the *dual* of our usual notion of an automaton, which accepts any string beginning in an initial state and ending in a final state (Hopcroft & Ullman, 1979). Instead, processes accept any string beginning in an initial state and ending in a non-final state.[4] A string is a sequence of actions (atoms). Therefore, by specifying the set of final states, we can infer the set of action sequences not permitted by the plan. It consists of those strings ending in a final state. All other action sequences that begin in an initial state are permitted by the plan. Processes are used here to be consistent with the automata theoretic verification literature.

Formally, a process is a three-tuple $S = (M_{\mathcal{K}}(S), I(S), F(S))$ where $\mathcal{K}$ is the Boolean algebra corresponding to $S$. $M_{\mathcal{K}}(S) : V(S) \times V(S) \rightarrow \mathcal{K}$ is the matrix of transition conditions, which are elements of $\mathcal{K}$, $V(S)$ is the set of vertices of $S$, $I(S) \subseteq V(S)$ are the initial states, and $F(S) \subseteq V(S)$ are the final states. Also, $E(S) = \{e \in V(S) \times V(S) \mid M_{\mathcal{K}}(e) \neq 0\}$ is the set of directed edges connecting pairs of vertices of $S$, and $M_{\mathcal{K}}(e)$ is the transition condition of $M_{\mathcal{K}}(S)$ corresponding to edge $e$. Note that we omit edges labeled "0." By our definition, an edge whose transition condition is 0 does not exist. We can alternatively denote $M_{\mathcal{K}}(e)$ as $M_{\mathcal{K}}(v_i, v_{i+1})$ for the transition condition corresponding to the edge going from vertex $v_i$ to vertex $v_{i+1}$. For example, in Figure 1, $M_{\mathcal{K}}$ (WAIT, GO) is (A: MAVs-wait) $*$ (C: go-A).

Figures 1 and 2 illustrate the process definitions. There are process plans for two agents: swarm A and task coordinator C. Recall that agent B is identical to A but with "A" replaced by "B." An incoming arrow to a state, not from any other state, signifies that this is an initial state. Recall that the output actions of process A are its atoms, and likewise for processes B and C. The transition conditions are the labels on the edges. We assume for process $X$ = A, B, or C, $F(X) = \emptyset$, i.e., there are no final states. Therefore every finite string of actions that starts in an initial state and satisfies the transition conditions is accept-

[4]For the case of deterministic and complete transition conditions, reversing the acceptance condition will complement the language.

able behavior for the plan.

A multiagent plan is formed from single agent plans by taking the *tensor product* of the processes corresponding to the individual plans. Essentially, this is done by taking the Cartesian product of the vertices and the intersection of the transition conditions. For details see Kurshan (1994). The product process models a set of synchronous processes. The Boolean algebra corresponding to the product process is the product algebra. For Figures 1 and 2, to formulate the process S modeling the entire multiagent plan, we take the tensor product S = A $\otimes$ B $\otimes$ C of the three processes. For this tensor product, $I$(S) = { (WAIT, WAIT, GO-A), (WAIT, WAIT, GO-B) }, and $F$(S) = $\emptyset$. The tensor product process is not shown in a figure because it's quite large.

Formally, a *string* $\mathbf{x}$ is a finite-dimensional vector, $(x_0, ..., x_n) \in \Gamma(\mathcal{K})^+$, i.e., a string is a sequence of one or more actions. A *run* $\mathbf{v}$ *of string* $\mathbf{x}$ is a sequence $(v_0, ..., v_{n+1})$ of vertices such that $\forall i$, $0 \leq i \leq n$, $x_i * M_{\mathcal{K}}(v_i, v_{i+1}) \neq 0$, i.e., $x_i \preceq M_{\mathcal{K}}(v_i, v_{i+1})$ because the $x_i$ are atoms.

The *language* of $S$ is $\mathcal{L}(S) = \{\mathbf{x} \in \Gamma(\mathcal{K})^+ \mid \mathbf{x}$ has a run in $M_{\mathcal{K}}(S)$ from $I(S)$ to $V(S) \setminus F(S)\}$. Such a run is *accepting.* The language of a plan is the set of all action sequences (i.e, strings) allowed by the plan.

An example string in the language of process S, the multiagent process that is the product of A, B, and C, is (((A:MAVs-wait) $*$ (B:MAVs-wait) $*$ (C:go-A)), ((A:MAVs-go) $*$ (B:MAVs-wait) $*$ (C:go-B)), ((A:MAVs-wait) $*$ (B:MAVs-go) $*$ (C:go-B)), ((A:MAVs-wait) $*$ (B:MAVs-go) $*$ (C:go-A))). This is a sequence of atoms of S. An accepting run of this string is ((WAIT, WAIT, GO-A), (GO, WAIT, GO-B), (WAIT, GO, GO-B), (WAIT, GO, GO-A), (GO, WAIT, GO-A)). Because $F$(S) = $\emptyset$, all runs beginning in an initial state are accepting runs and they form the elements of the language of S.

## 3.2 TEMPORAL LOGIC PROPERTIES

We assume properties are expressed in temporal logic. For formal versions of the definitions here, see Manna and Pnueli (1991). Linear time is assumed here. In other words, time proceeds linearly and we do not consider simultaneous possible futures. The type of verification used in this paper is "model checking." In other words, verification tests whether $S \models P$ for plan $S$ and property $P$, i.e., whether plan $S$ "models," or satisfies, property $P$.

For consistency with the temporal logic literature, we define a *computational state* (*c-state*) as the action chosen from each process state. Then a computation is a finite sequence of temporally ordered computational states, i.e., a string. To distinguish the two types of states, we will refer to a process state as a *p-state.*

$P$ is a property true (false) for a process $S$, i.e., $S \models P$ ($S \not\models P$), if and only if it is true for every string in the language $\mathcal{L}(S)$ (false for some string in $\mathcal{L}(S)$). The notation $\mathbf{x} \models P$ ($\mathbf{x} \not\models P$) means string $\mathbf{x}$ satisfies (does not satisfy) property $P$, i.e., the property holds (does not hold) for $\mathbf{x}$. Before defining what it means for properties to be true (i.e., hold) for a string, we first define what it means for a formula that is Boolean expression to be true at a c-state. A *c-state formula* $p$ is true (false) at c-state $x_i$, i.e., $x_i \models p$ ($x_i \not\models p$) if and only if $x_i \preceq p$ ($x_i \not\preceq p$), i.e., $x_i * p \neq 0$ ($= 0$) because $p$ is a Boolean expression with no variables on the same Boolean algebra used by process $S$, and $x_i$ is an atom of that algebra. For example, (A:MAVs-wait) $\models$ ((A:MAVs-wait) + (A:no-MAVs)) for c-state (A:MAVs-wait) and c-state formula ((A:MAVs-wait) + (A:no-MAVs)).

A c-state formula $p$ is true/false in particular c-states of a string. Property $P$ is defined in terms of $p$, and is true/false of an entire string, i.e., $\mathbf{x} \models P$ or $\mathbf{x} \not\models P$ for string $\mathbf{x}$. We now define two property classes that are among those most frequently encountered in the verification literature for finite strings. Assume $\mathbf{x} = (x_0, ..., x_n)$ is a string of process $S$. For c-state formula $p$ and plan $S$, define Sometimes property $P = \Diamond\, p$ ("Sometimes $p$") as a property that is true for string $\mathbf{x}$ if only if $p$ is true in at least one c-state $x_i$ of $\mathbf{x}$, where $0 \leq i \leq n$. An Invariance property $P = \Box p$ ("Invariant $p$") is a property true for string $\mathbf{x}$ if and only if $p$ is true in every c-state $x_i$ of $\mathbf{x}$.

Continuing with the MAVs example, a desirable Invariance property $P_I$ states that "only one MAV enters the region at a time." This can be expressed in temporal logic as $P_I = \Box(\neg$ ((A:MAVs-go) $*$ (B:MAVs-go))). A desirable Sometimes property $P_S$ states that "Sometimes MAVs from swarm A enter the region." In logic this property is expressed as $P_S = \Diamond$ (A:MAVs-go). $P_I$, but not $P_S$, holds for the multiagent plan S.

## 3.3 "SAFE" LEARNING

This paper is concerned with *"safe" machine learning* methods *(SMLs)*, i.e., machine learning operators that preserve properties, also called "correctness preserving mappings." For plan $S$ and property $P$, suppose verification has succeeded prior to learning, i.e., $\forall \mathbf{x}$, $\mathbf{x} \in \mathcal{L}(S)$ implies $\mathbf{x} \models P$ (i.e., $S \models P$). Then according to Gordon (1997a), a machine learning operator $ml(S)$ is an SML if and only if verification succeeds after learning, i.e., $\forall \mathbf{x}$, $\mathbf{x} \in \mathcal{L}(ml(S))$ implies $\mathbf{x} \models ml(P)$. Note that a machine learning operator may also affect the property $P$, which could be undesirable. Therefore, being an SML is not always sufficient. Additional requirements on learning – in particular, abstraction, are discussed next.

## 4 BOOLEAN ALGEBRA ABSTRACTION

Kurshan (1994) presents methods for improving the efficiency of automata-based verification, but does not consider the possibility of automata, such as agents, that can learn. By applying some of the results of Kurshan (1994) in a novel way, Gordon (1997a; 1997b) shows that when agents learn using certain abstractions, the abstractions are a priori guaranteed to be SMLs for *all* property classes – but *only* if abstraction is performed to both the plan and property. [5] Therefore, this section identifies situations in which it is acceptable to apply an SML abstraction to a property.

The SML abstractions include very useful ones, such as partitioning the Boolean algebra atoms e.g., using constructive induction, and projection, which is a form of feature selection (or, more properly, action deletion). Although the methods described in this section apply to any of these abstractions, for illustration we focus only on projection, which is a mapping from a Boolean algebra to a subalgebra. For a formal definition of projection, see Kurshan (1994). Here, we continue with the MAVs example.

Suppose all the MAVs in the B swarm leave the region. To incorporate this knowledge, Boolean algebra projection, a type of abstraction, projects the product algebra $\mathcal{ABC}$ onto subalgebra $\mathcal{AC}$. Projection $proj_{\mathcal{AC}}$ : $\mathcal{ABC} \rightarrow \mathcal{AC}$ is defined as $proj_{\mathcal{AC}}(a * b * c) = a * c$ for atoms $a \in \Gamma(\mathcal{A})$, $b \in \Gamma(\mathcal{B})$ and $c \in \Gamma(\mathcal{C})$, and is extended linearly to the full algebra. For example, $proj_{\mathcal{AC}}$ ((A: MAVs-wait) $*$ (B: MAVs-wait) $*$ (C: go-A)) = (A: MAVs-wait) $*$ (C: go-A). In addition to removing entire subalgebras, it is also possible to remove atoms from within a subalgebra.

Projection $proj_{\mathcal{AC}}$ removes references to swarm B from the multiagent plan S. It therefore eliminates the need for multiagent coordination with swarm B. We assume that when the agent applies a projection to the plan, it has justification to do so – because the purpose of abstraction is to modify the plan. Modification of the property, on the other hand, may be a side effect required for an a priori guarantee that the abstraction is an SML. Applying $proj_{\mathcal{AC}}$ to the Invariance property $P_I$, which states that "only one MAV may enter the region at a time," results in a property which accepts *any* multiagent plan of agents A, B, and C. When applied to both plan and property, $proj_{\mathcal{AC}}$ is an SML. Nevertheless, if the B swarm returns to the region and is restored into the multiagent plan, then this new property which allows the agents to do anything could have disastrous, unintended (by the user) consequences.

This example illustrates our dilemma: If we abstract the property along with the plan, the abstraction will be guaranteed a priori to be an SML. However, by abstracting the property, we risk violating the user's original intentions. *When is it ok to abstract a property?* There are at least three cases when it is permissible:

(1) When the abstraction is *property invariant.*

Applying the projection $proj_{\mathcal{AC}}$ to the Sometimes property $P_S$, which states that "Sometimes MAVs from swarm A enter the region," leaves $P_S$ invariant, i.e., $proj_{\mathcal{AC}}(P_S) = P_S$. Therefore the abstraction is property invariant. The intuition is that the behavior of agent B is irrelevant when testing this property.

In general, to determine whether property invariance holds, an agent must apply abstraction to each property $P$ and then check whether $P$ remains unaltered by abstraction. This simple syntactic check is a form of incremental re-verification because it is localized to a test on the property alone. The check has a worst case time complexity of $O(|P|)$ for any property $P$. This is lower than the worst case time complexity of complete re-verification from scratch (following abstraction), which is $O(|\Gamma(\mathcal{K})| * |P|)$ for Invariance and Sometimes properties, where $|\Gamma(\mathcal{K})|$ is the number of atoms in the plan ( Lichtenstein & Pnueli, 1984). Furthermore, if the agent will only accept property invariant abstractions, then the cost of plan abstraction can be avoided when this incremental check fails.

(2) When the abstraction is *property irrelevant.*

An example is when the agents discover, or are told about, a *permanent* change that henceforth renders one or more items (e.g., an agent or action) irrelevant. The term "permanent" in this context means a change

[5] This result applies to agents with finite or infinite lifetimes.

whose effects are sustained at least until the last agent has terminated. Because the change is permanent, we can be assured that no problems are caused by applying an SML abstraction to the properties.

Consider an example in which a swarm agent becomes irrelevant. Suppose the lives of all MAVs in the B swarm have terminated, e.g., they become permanently inoperative, but we wish to continue with the multiagent plan because the other agents survived. Then the application of $proj_{\mathcal{AC}}$ to the property $P_I$ has no significant effect – because $P_I$ is no longer needed.

(3) When the abstraction is *property reversible*.

Suppose the agents determine that one or more items are not relevant to the objectives of their multiagent plan, but this is a *temporary* change in condition, i.e., the items may become relevant again. For example, agents may disappear to attend to other tasks then possibly return, and actions may become temporarily disabled due to mechanical failures. Items irrelevant to the multiagent objectives could be removed from the multiagent plan and also from the properties. In these circumstances, we want the abstraction to be property reversible. An abstraction is property reversible if the pre-abstraction property can be restored, e.g., by saving it. This way we can retest the original property after undoing the effects of abstraction.

We only want our agents to perform property irrelevant and property reversible abstractions when abstraction is restricted to removing irrelevant items. If agents are not told relevance, they may need to perform relevance determination, perhaps using methods such as those of Subramanian (1988). Other research related to the ideas in this section includes feature selection (see http://ai.iit.nrc.ca/bibliographies/feature-selection.html), and plan abstraction (Knoblock, 1990).

## 5 GENERALIZATION

Although we have been unable to obtain positive a priori results for generalization, this section presents a novel method for incremental re-verification after generalization. Efficiency is gained by tailoring incremental re-verification methods to specific property classes. Because there are only about a dozen property classes commonly used in practice (Kurshan, 1994), this seems reasonable to do. The re-verification method presented in this section is specific to Invariance properties. A method for Sometimes properties may be found in Gordon (1997b). Methods for other property classes are currently being investigated.

Generalization differs from abstraction in that you are not changing the entire Boolean algebra (e.g., taking a subalgebra) but instead you are increasing the generality of a transition condition labeling one or more edges (for simplicity, here we consider one). Generalization is done when the agent discovers that the transition can/should be taken under a larger set of circumstances. It is only done to the plan. In the context of a process, generalization raises the level of a particular p-state-to-p-state transition condition in the partial order $\preceq$, whereas specialization lowers it, e.g., as in Mitchell's Version Spaces (Mitchell, 1978).

Formally, we define generalization of the condition along edge $(v, w)$ as follows. Generalization operator $ml_{gen} : S \rightarrow S'$, where both $S$ and $S'$ use Boolean algebra $\mathcal{K}$, is defined as $ml_{gen} : M_{\mathcal{K}}(S) \rightarrow M_{\mathcal{K}}(S')$, where $ml_{gen}(M_{\mathcal{K}}(v, w)) = M_{\mathcal{K}}(v, w) + z$, for some $z \in \mathcal{K}$.[6]

An example of generalization is the following. The transition condition associated with the edge ((WAIT, WAIT, GO-A), (GO, WAIT, GO-B)) in the multiagent plan S is (A:MAVs-wait) $*$ (B:MAVs-wait) $*$ (C:go-A). This could be generalized to ((A:MAVs-wait) $*$ (B:MAVs-wait) $*$ (C:go-A)) + ((A:MAVs-wait) $*$ $\neg$(B:MAVs-wait) $*$ (C:go-A)), i.e., (A:MAVs-wait) $*$ (C:go-A) for new plan S'.

To illustrate our incremental approach, recall S satisfies the Invariance property $P_I$ which states that "only one MAV enters the region at a time," i.e., $\Box$ ( $\neg$ ((A:MAVs-go) $*$ (B:MAVs-go))). We could check this property against the entire, new plan S', but a preferable alternative is to simply check it against the new addition to the transition condition, namely, is $P_I$ satisfied by (A:MAVs-wait) $*$ $\neg$ (B: MAVs-wait) $*$ (C:go-A)? In fact it is, because (A:MAVs-go) is not true, and that is all we need to know to be sure that the $ml_{gen}$ just applied is an SML. We can now formalize this.

Let us consider the Invariance property $P = \Box\, p$ for c-state formula $p$. Let $y$ be the existing transition condition for edge $(v, w)$ in plan $S$, i.e., $M_{\mathcal{K}}(v, w) = y$. We previously defined what it means for a c-state formula $p$ to be true at a c-state, but it is also useful to define what it means for a c-state formula to be true of a transition condition. Let $\Gamma(\mathcal{K})_y = \{a \mid a \in \Gamma(\mathcal{K})$ and $a \preceq y\}$. A c-state formula $p$ is defined to be true of a transition condition $y$, i.e., $y \models p$, if and only if $\forall a \in \Gamma(\mathcal{K})_y$, $a \preceq p$.

Assume every string $\mathbf{x}$ in $\mathcal{L}(S)$ satisfies Invariance

[6] $S'$ differs from $S$ only by the results of $ml_{gen}$.

property $P$, so for each $\mathbf{x}$, $p$ is true of every atom in $\mathbf{x}$. This implies $y \models p$.[7] Now we generalize the edge $(v, w)$ to form $S'$ via $ml_{gen}$ $(M_{\mathcal{K}}(v, w)) = y + z$. This operator $ml_{gen}$ is an SML with respect to Invariance property $P$ if and only if $S' \models P$, which is true if and only if $z \models p$. The reason for this is that we know $S$ satisfies $P$ from our original verification, and therefore $p$ is true for all atoms in all strings in $\mathcal{L}(S)$. The only new atoms in $\mathcal{L}(S')$ but not in $\mathcal{L}(S)$ are in $\Gamma(\mathcal{K})_z$. Therefore, if $z \models p$, then $p$ is true for all atoms in $\mathcal{L}(S')$, which implies every string in $\mathcal{L}(S')$ satisfies $P$, i.e., $S' \models P$. Therefore, re-verification need only test whether $z \models p$, i.e., $\forall a \in \Gamma(\mathcal{K})_z$, $a \preceq p$. (We assume transition conditions are represented extensionally, i.e., as the unique sum of atoms equivalent to the Boolean expression.) If $z \not\models p$, $S' \not\models P$.[8] This test is *incremental* because it is localized to just checking whether the property holds of the newly added atoms in $z$, rather than all atoms in $\mathcal{L}(S')$.

For example, suppose a, b, c, d, and e are atoms, and the transition condition $y$ between $v$ and $w$ equals a. Let (a, b, b, d) be an accepting string of $S$ that includes $v$ and $w$ as the first two vertices in its accepting run. The property is $P = \Box \neg$ e. Assume the fact that this string satisfies $\neg$ e was proved in the original verification. Suppose $ml_{gen}$ generalizes $M_{\mathcal{K}}(v, w)$ from a to (a + c), which adds the string (c, b, b, d) to $\mathcal{L}(S')$. Then rather than test whether the elements of { a, b, c, d } are $\preceq \neg$ e, all we really need to test is whether c $\preceq \neg$ e – because c is the only newly added atom.

By storing and reusing knowledge from previous verification(s), we can increase the efficiency of this test. Suppose some atoms $a$ such that $a \preceq z$ were tested for $a \preceq p$ during previous verification(s), and the outcomes of these tests were stored. Then lookup will suffice, and the only atoms in $\Gamma(\mathcal{K})_z$ that need to be tested against $p$ during the current re-verification are those not previously tested.

What cost benefit(s) does incremental re-verification have over complete re-verification from scratch? Verification, or complete re-verification from scratch, in the worst case has time complexity $O(|\Gamma(\mathcal{K})| * |p|)$ for Invariance properties, where $|\Gamma(\mathcal{K})|$ is the total number of atoms, and $|p|$ is the length of the c-state formula $p$ (Lichtenstein & Pnueli, 1984). This is because the c-state formula may have to be tested in every unique c-state, which is an atom. $|\Gamma(\mathcal{K})|$ is exponential in the number of single agent plans forming a multiagent plan. In the worst case, incremental re-verification has the same time complexity, but this would be a very bizarre situation indeed. It would require that no atoms were tested against the property in the original verification (which could occur if $\mathcal{L}(S)$ were empty), and *all* atoms are added to the transition condition during generalization, i.e., $\forall a \in \Gamma(\mathcal{K})$, $a \preceq z$.

Let us consider a more realistic comparison. The worst case time complexity for complete re-verification assumes all c-states are reachable from some initial p-state. This may not be true, e.g., the number of initial p-states might be very small. Re-verification is required to determine $\forall \mathbf{x} \in \mathcal{L}(S')$ whether $\mathbf{x} \models P$. At the very least, complete re-verification of an Invariance property $P = \Box p$ must test whether $x_i \models p$ $\forall x_i$ in $\mathbf{x}$, $\forall \mathbf{x} \in \mathcal{L}(S')$. The complexity of this test is $C_{complete} = O(|\Gamma(\mathcal{K})_{\mathcal{L}(S')}| * |p|)$, where $|\Gamma(\mathcal{K})_{\mathcal{L}(S')}|$ is the number of unique atoms in all strings $\mathbf{x} \in \mathcal{L}(S')$.

A more realistic cost estimate for incremental re-verification is $C_{increm} = O(|\Gamma(\mathcal{K})_{s(z)}| + (|\Gamma(\mathcal{K})_{ns(z)}| * |p|))$, where $\Gamma(\mathcal{K})_{s(z)}$ $(\Gamma(\mathcal{K})_{ns(z)})$ contains atoms whose results are (are not) previously stored. The first addend is the cost of lookup of results from previous verification(s), and the second addend is the cost added by testing the atoms that were not previously tested. Whenever generalization is reasonably conservative, i.e., $|\Gamma(\mathcal{K})_z| << |\Gamma(\mathcal{K})_{\mathcal{L}(S')}|$, incremental can provide considerable savings over complete re-verification!

## 6 DISCUSSION

Here we have addressed the question of how agents can adapt (learn) safely, i.e., by preserving critical properties, and how they can do this in a time-efficient manner. We extended the work of Gordon (1997a) to obtain positive results for two popular machine learning methods: abstraction and generalization. For abstraction to be a priori safe (property-preserving), the property must also be abstracted. This paper enumerates situations in which it is permissible to abstract the property. Furthermore, novel incremental re-verification methods are presented for abstraction and generalization. These methods have the potential to provide large computational savings over complete re-verification from scratch. With our methods (including a priori), agents can use abstraction and generalization to adapt to novel situations, and can do so with quick checks that ensure the reliability of their behavior.

[7]This statement is based on our assumption that $(v, w)$ is part of an accepting run for at least one $\mathbf{x} \in \mathcal{L}(S)$. This assumption motivates re-verification.

[8]That is, unless $(v, w)$ is not part of any accepting run – but then the test is unnecessary.

There is a small amount of prior research on incremental re-verification. Reps and Teitelbaum (1989) developed a verifier for users to check their code while writing in traditional programming languages, such as PL/I. Their verifier can incrementally re-check software after edits using Hoare-style proofs. However, unlike our re-verification methods, these proofs require some interaction with the user. Sokolsky and Smolka (1994) have an incremental method for verifying added or deleted state transitions in an automaton-like representation. However they do not address generalization or abstraction. Finally, Weld and Etzioni (1994) have a method to incrementally test an agent's plan to decide whether to add new actions to the plan. There are certain similarities between our work and that of Weld and Etzioni. They add actions to a plan only when their effects do not violate *dont-disturb* properties, which are a type of Invariance property. Our generalization also adds actions to a plan. Furthermore, both approaches localize verification. The main differences are that unlike Weld and Etzioni, we: (1) use a formal foundation based on the verification literature, in particular, model-checking and automata, (2) assume the existence of prior verification knowledge and use this knowledge to streamline re-verification, (3) use reactive rather than necessarily goal-oriented plans, and (4) address abstraction.

One aspect of Weld and Etzioni (1994) that was purposely not addressed here is that of how to select which method to use in repairing a plan. This is a rich issue for future research, and could draw on cost-effective methods such as those of Joslin and Pollack (1994). Rather than repair, this paper focuses on re-verification. We are unaware of any methods besides ours for incrementally re-verifying abstraction or generalization in automata. Much more work remains to be done on the important topic of incremental re-verification – especially for adaptive agents.

## Acknowledgments

Thanks to Bill Spears and Sampath Kannan for useful inputs on this paper. Bill suggested the alliterative title. This research was sponsored by the Office of Naval Research N001498WX20296.

## References

Asimov, I. (1942). Runaround. In *Astounding Science Fiction.*

Gordon, D. (1997a). Asimovian adaptive agents. NCARAI Technical Report 97-016.

Gordon, D. (1997b). Machine learning and finite-lifetime agents: Some preliminary results. NCARAI Technical Report 97-017.

Hopcroft, J. & Ullman J. (1979). *Introduction to Automata Theory, Languages, and Computation.* Menlo Park: Addison-Wesley.

Joslin, D. & Pollack, M. (1994). Least-cost flaw repair: A plan refinement strategy for partial-order planning. *Proceedings of AAAI94* (pp. 1004-1009). AAAI Press.

Knoblock, C. (1990). A theory of abstraction for hierarchical planning. In D. P. Benjamin (Ed.), *Change of Representation and Inductive Bias.* Norwell: Kluwer Academic Publishers.

Kurshan, R. (1994). *Computer Aided Verification of Coordinating Processes.* Princeton, N.J.: Princeton University Press.

Laumer, K. (1976). *Bolo, The Annals of the Dinochrome Brigade.* New York, N.Y.: Berkeley Publishing Corp.

Lichtenstein, O. & Pnueli, A. (1984). Checking that finite state concurrent programs satisfy their linear specifications. *Proceedings of the Twelfth ACM Symposium on Principles of Programming Languages* (pp. 271-276).

Manna, Z. & Pnueli, A. (1991). Completing the temporal picture. *Theoretical Computer Science, 83(1),* 97-130.

Mitchell, T. (1978). *Version Spaces: An Approach to Concept Learning.* Ph.D. thesis, Stanford University.

Reps, T. & Teitelbaum, T. (1989). *The Synthesizer Generator.* New York: Springer-Verlag.

Saberhagen, F. (1967). *Berserkers.* New York: The Berkley Publishing Group.

Sokolsky, O. & Smolka, S. (1994). Incremental model checking in the modal mu-calculus. *Proceedings of Computer-Aided Verification.*

Subramanian, D. (1988). *A Theory of Justified Reformulations.* Ph.D. thesis. Stanford University.

Weld, D., & Etzioni, O. (1994). The First Law of Robotics. *Proceedings of AAAI94* (pp. 1042-1047). AAAI Press.