Towards Safe Learning Agents

Mike Barley and Hans Guesgen
Computer Science Dept., University of Auckland
Private Bag 92019, Auckland, New Zealand
Phone: +64 9 373 7599, Fax: +64 9 373 7453
{barley, hans}@cs.auckland.ac.nz

ABSTRACT

As intelligent software agents are pushed out into the real world and increasingly take over responsibility for important tasks, it becomes important that there are ways of guaranteeing that the agents are safe. Guaranteeing that an agent is safe involves more than simply guaranteeing that its plans are safe. It also involves guaranteeing that the agent can find adequate solutions to its target set of problems in a timely manner. In other words, the agent is safe with respect to its desired coverage, speed, and solution quality. Given such a safe agent, we want to guarantee that no learning done by the agent makes it unsafe. In this paper we present one approach to guaranteeing such a result and illustrate it with an abbreviated case study.

Categories and Subject Descriptors

I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search

General Terms

Languages

Keywords

learning, agents, planning, safety

1. INTRODUCTION

Some tasks that are too costly, too demanding, and/or too boring there is a push to get intelligent software agents out into the real world. One example is NASA's Remote Agent experiment (RAX)[7], which controlled NASA's Deep Space 1 (DS1)[11] spacecraft for two days in May, 1999. There is also a push to get intelligent software agents out onto the internet as intelligent information gatherers.

Along with this desire to get intelligent software agents out in the real world, there has been concern[12] voiced that agents might cause harm to people. Perhaps HAL in Kubrick's "2001: A Space Odyssey" is the best image of such an agent gone wrong. An agent might go "wrong" in a number of different ways: harmful plans, harmful planning behaviors, harmful plan execution, or harmful learning.

An agent's plans could be harmful by containing actions that were individually harmful or harmful in combination. An agent's planning behavior could be harmful in situations where the planner could not come up with an adequate plan within the required time. An agent's plan execution could be harmful if it did not monitor its environment adequately or if the precision of its actions was less than demanded by the plan. Finally, the agent's learning could lead to any of the three preceeding problems.

While there has been concern about safe agents, there was been little research in this area. In general, it is probably impossible to prove that an agent is safe and continues to be safe as it learns. However, we need to find out what guarantees are possible. Most of the current research has focused on guaranteeing that the agent produces safe plans and the effect of the agent's learning upon these guarantees. However, this paper focuses on what conditions are needed to guarantee that certain aspects of the agent's planning behavior is still be safe after learning occurs.

1.1 Safe Plans

Weld and Etzioni[12] discuss the use of "don't-disturb" safety constraints that constrain an agents plans by describing properties that no state is allowed to have as a result of executing that plan. An example of a safety constraint in the Unix world might be that the agent is not allowed to delete files that haven't been backed-up. If the planner can only create plans that satisfy these safety constraints then the planner should be relatively safe[1].

Gordon[6] tackles a slightly different problem. Gordon's agents have plans to handle specific tasks. Weld & Etzioni's planner creates plans for a specific initial state and goal pair, where the plan is guaranteed to be safe. However, Gordon's agent's plans are expected to be applied to unanticipated states where the plan may need to be perturbed to become applicable. Proving that these plans along with their possible perturbations are safe is a much more difficult problem. This problem requires formal verification that regardless of the initial state and perturbations, that the states resulting applying that plan still satisfy the safety conditions. Moreover, Gordon is specifically interested in the case where the agent is actually learning new ways to perturb its plans and

[1]Plans can never be perfectly safe, agents have imperfect models of domain actions and of the world.

AGENTS-2001 Workshop on LEARNING AGENTS 2001 Montreal, Canada

Gordon want to reverify that even with this new knowledge the safety conditions are guaranteed to hold.

1.2 Safe Agents

Both Weld/Etzioni and Gordon focus on ensuring that plans are safe. However, simply guaranteeing an agent's plans are safe is not enough to guarantee the agent is safe. In general, we want assurances that the agent can produce solutions to all problems in its target problem class, within its time allowance, and of acceptable quality. These three comprise the planner's performance dimensions: coverage, speed, and solution quality. These performance assurances are usually accomplished through some combination of formal analysis and/or practical testing. While we want our agents to learn from their environment, we don't want to redo the entire quality assurance process all over again.

An agent cannot be called safe if it cannot come up with solutions for its target problems. For example, if we have an agent controlling a space shuttle, one of its target set of problems may be docking with the space station. If the agent cannot handle a normal docking situation then it is not safe for that agent to control the docking maneuver. If the agent takes so long to come up with a docking plan that it is no longer applicable or if the plan is so poor that docking is so rough that equiment breaks, etc., then it is not safe for that agent to be responsible for solving this type of problem. We call an agent safe if its plans satisfy Weld/Etzioni's safety constraints and its performance is guaranteed to lie within its target values for its main performance dimensions (e.g., coverage, speed, and/or solution quality).

1.3 Safe Learning Agents

Many agents, including RAX, use search control rules (SCRs) to help them plan effectively in their environments. We want our agents to learn their search control rules. Given a planning algorithm and an initial set of search control rules such that the agent is safe and given a particular learning algorithm we like to be able to guarantee that after learning the system is still be safe. To be able to avoid re-doing the quality assurance process all over again each time the agent learns a new search control rule, we want the planner to satisfy an inductive safety preservation property. In other words, given a set of search control rules for which we already know (from our quality assurance process) that the system is safe and given a new search control rule for which we can determine that the system is safe, we want to gurantee that for the search control rule set resulting from adding the new rule, that the system is still safe.

If the planner has this inductive safety preservation property and if we can prove that the learner only learns search control rules that individually are safe for the system, then we have a guarantee that if the initial set of search control rules are safe for the planner then all future rule sets resulting from adding in learned rules are also safe for the planner. This means that no additional resources need be expended to verify the safety of the agent.

1.4 Paper's Focus

We assume the initial safety assurance can be carried out via traditional means. We want to avoid redoing the assurance process each time the agent learns. In this paper the agent's main performance dimension is its target problem coverage. We distinguish between the potential performance of an agent and its actual performance. The first is reflected in the search space topology, whereas the latter is determined by the traversal of the search space. We focus here on the topology.

Barley&Guesgen[2] shows how the use of meta-level preconditions in their SCRs can lead to counter-intuitive behavior in the following problem-solvers: Prodigy 2.0[9], Prodigy 4.0[4], Soar[8], and UCPOP[3]. More specifically, that adding a safe SCR to a safe SCR rule set can lead to an unsafe agent. From now on, Prodigy refers to Prodigy 2.0.

This paper presents an abbreviated case study of a learning agent, where we illustrate this problem and discuss the conditions that guarantee an inductive coverage preservation (ICP) property for the agent. The case study looks at the Prodigy planner and the Bacall[1] learner.

2. CASE STUDY: PRODIGY/BACALL

Bacall[1] is the only SCR learner for which conditions have been identified for when the search control rules that it learns cannot decrease its problem-solver's coverage. This was done by identifying the conditions (the *ICP conditions*) that guarantee the ICP property for Prodigy and then showing that under these ICP conditions adding Bacall learned rules never remove solutions from Prodigy's search space. We use this as a case study of describing the ICP conditions under which a specific problem-solver/learner system is a safe learning agent with respect to coverage.

We first look at some of the features of its problem-solver, Prodigy, then look at the type of learning done by Bacall. We then identify which features prevent Prodigy/Bacall from having any general guarantees of inductive coverage preservation. Finally, we discuss the ICP condition that guarantee that Prodigy/Bacall is a safe learning agent.

2.1 Prodigy's Search Control Executive

Prodigy is a classical means-ends planner. Nodes in Prodigy's search space represent operator subgoaling and operator application actions and contain their current goal stack and state. Operator subgoaling pushes operators and their sets of unsatisfied preconditions onto the goal stack. Operator application pops the operator at the top of the goal stack and uses that operator's add and delete lists to update the current state. Operator application occurs automatically whenever an applicable operator appears on the top of the goal stack. Operator subgoaling is a decision process that goes through a number of phases which decide what operator to push onto the goal stack. The first phase decides which goal to pursue next, the second decides which operator to use to achieve that goal, and the third finishes instantiating any unbound operator parameters.

Each phase goes through a sequence of subphases: candidate generation, selection, rejection, and finally ordering. The set of candidates for a phase are created during the generation phase, and passed through to and updated by each successive subphase. The generation subphase is done implicitly by the Prodigy code. Each of the other subphases has Prodigy defaults that can be augmented by explicit search control rules. The selection subphase was intended to select candidates that were thought to be guaranteed to succeed while implicitly rejecting all non-selected candidates. By default, no candidates were "selected". However, the selection code did not actually check that the "selected" candidate had actually been generated by Prodigy, thus selection search control rules could actually augment the set of candidates available to the later subphases. The rejection subphase was intended to identify those candidates that should be removed from the set of current candidates. By default, if any candidates had been "selected" then all non-selected candidates were rejected. The last subphase, ordering, determined in which order the candidates should be explored. This paper is not concerned with node traversal order.

For Prodigy/Bacall, the target set of problems are those for which Prodigy's search space (without explicit search control rules) contains solutions. Prodigy's candidate generation code implemented a number of search heuristics: strong linearity[5], state loop rejection, goal loop rejection, etc. Informally, strong linearity (SL) forbids subplans for different goals to be interleaved. This restriction sometimes prevents Prodigy from being able to solve otherwise solvable problems. Except for strong linearity, the rest of the heuristics turn out not to be important for our discussion, so we do not mention them further.

2.2 Description of Bacall Learning

Because Prodigy uses the SL heuristic, for certain problems it is possible that Prodigy's implicit rejection heuristics remove an edge from every path to every solution. Bacall is given a problem description (e.g., initial state and goals) and a solution. If strong linearity is preventing Prodigy from finding a solution then Bacall learns subgoal generation search control rules which proposes the appropriate subgoal at the correct point during plan generation to put that edge back into Prodigy's search space.

Bacall learns this type of rule by searching for SL violations in the plan. When Bacall detects an SL violation it determines which goals' plans were interleaved and at what plan step the interleaving occurs. If those goals cannot be achieved without interleaving their plans then Bacall uses Prodigy's EBL[10] module to compute the explanation for that failure. That explanation becomes the preconditions and generation of the subgoal achieved by the interleaved step becomes the postcondition. Achieving this promoted goal allows Prodigy to then achieve the remaining goals.

2.3 Analysis of Prodigy's ICP Problem

Because Bacall learned rules, individually, only add edges to the search space, we'd like to be able to guarantee that Prodigy has the ICP property. Unfortunately, this is not the case. This is easy to demonstrate. All we need to show is that there exists a problem, X, where the given search control rule set, RS, preserves the solutions and the new search control rule, S, preserves its solutions, but the rule set resulting from adding S to RS does not preserve its solutions. To do this all we need to show is a solution for X, that RS and S preserve, but which $RS \cup \{S\}$ does not.

Table 1 shows a problem, X, with one solution: OP1. Figure 1(a) shows for the given SCRs the effect of the different subphases for the operator decision phase after the goal decision phase has finished. Prodigy's default logic creates one candidate operator (OP1) and the selection subphase selects it. The rejection subphase does not reject OP1 because the reject SCR's precondition requires both OP1 and OP2 to be candidates to fire. Now, because OP1's preconditions are satisfied, Prodigy solves the problem by applying OP1.

Figure 1(b) shows what happens after the new select SCR S is added to the rule set. Prodigy's default logic still creates

Table 1: Example of Prodigy's Lack of ICP Property

```
Domain Operators
(OP1 (preconds (C1))
     (effects ((add G1))))
(OP2 (preconds (C1))
     (effects ((add G2))))

Given SCRs
IF: (AND (CURRENT-NODE <N>)
         (CANDIDATE-OP <N> OP1)
         (CANDIDATE-OP <N> OP2))
THEN: (REJECT OPERATOR OP1)
IF: (AND (CURRENT-NODE <N>)
         (CANDIDATE-OP <N> <OP>))
THEN: (SELECT OPERATOR <OP>)

New SCR S
IF: (AND (CURRENT-NODE <N>)
         (KNOWN <N> C1))
THEN: (SELECT OPERATOR OP2)

Problem Definition
Initial State: ((C1))
Goal: (G1)
```

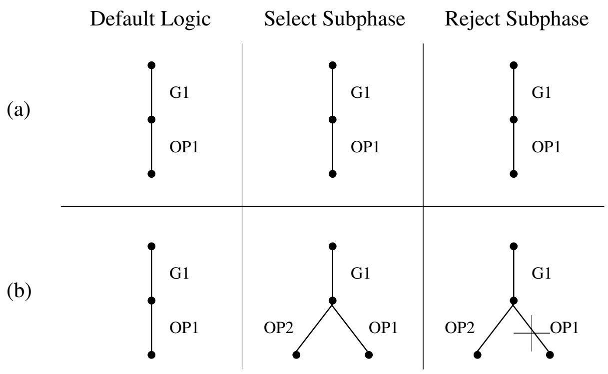

Figure 1: Effect of the subphases for the operator decision phase (a) without S and (b) with S.

the one candidate operator (OP1). However, now the select subphase not only selects OP1, it also selects OP2. This means that in the reject subphase, the preconditions for the reject rule are satisfied and operator OP1 is rejected. Now, because OP2's preconditions are satisfied, Prodigy applies OP2 but the goal has not been achievd. In fact, Prodigy repeats this process indefinitely.

From the example, the source of the problem should be clear. The problem is Prodigy's SCR can have meta-level preconditions which query Prodigy's problem-solving state. In particular, an SCR precondition can query which alternatives are available at a choice point. This means that the search control rules are sensitive to what is happening in the problem-solver's meta-level (e.g., the alternatives available at a search space node as in "(CANDIDATE-OP <N> OP1)"). This means that SCR rules that add or remove alternatives from choice points affect what alternatives are created/removed at other choice points.

2.4 Solution to Prodigy/Bacall's ICP Problem

Because Bacall learned rules only add edges to the search space, we'd like to be able to state that adding a Bacall learned search control rule to the current rule set can never cause edges in Prodigy's search space to be removed. However, as we have just seen, simply adding a "generate" search control rule to a rule set can cause the search space to actually decrease. We now state an ICP condition which if the current set of SCRs satisfy and if the new Bacall rule satisfies then adding that rule to the current rule set will not cause edges to be removed.

The ICP condition is that the SCR preconditions only query the state[2] of nodes directly on the path from the search tree's root to the alternative mentioned in the postcondition.[3] Given a node, N, in the search tree, adding an edge to the search tree will not change the state of any node along the path from the root to N. Since SCR preconditions can only query the state of the nodes along that path, adding nodes elsewhere in the tree will not affect the preconditions for any rule that might reject N.

To show that Prodigy/Bacall has the ICP property, we now only need to show that Bacall-learned rules are always safe and always satisfy Prodigy's ICP condition. Since Bacall rules only add edges to the tree, they are individually safe with respect to coverage. Bacall uses Prodigy's EBL component to compute the explanation for why Prodigy cannot solve the training problem with its current set of search control rules. This explanation is the basis for the preconditions for Bacall's rules. Examination of Prodigy's code reveals that those preconditions only reference the current node (goal, operator, and/or binding), goals pending in the current goal stack, and conditions true in the current world state. Therefore the explanations, and consequently Bacall's rule preconditions, only query the state of the nodes along the path from root to the node being "generated" by that rule. Thus, Bacall-learned rules are guaranteed to both be individually safe and to satsify Prodigy's ICP condition. Consequently, the ICP property holds for Prodigy/Bacall. That is, if Prodigy's initial set of SCRs are safe and satisfy its ICP conditions then adding Bacall learned rules always results in a safe set of rules that satisfy the ICP conditions.

3. CONCLUSIONS

Intelligent software agents are beginning to be pushed out into the real world to assume responsibility for performing important tasks. It is clear that before we give those agents responsibility for making and implementing important decisions, we need to feel relatively confident that it is safe to give those agents that responsibility. Simply guaranteeing that the agent's plans are safe is not enough. Specifically, an agent has to be safe with respect to its performance dimensions: problem coverage, planning speed, solution quality, and plan execution monitoring and control. Unfortunately, it is both difficult and costly to achieve any assurance that the agent is indeed safe with respect to these dimensions.

Many intelligent software agents also learn. In these cases, it is not enough that the agent is safe when it goes "live", it needs to remain safe as it continues to learn and adapt to its environment. However, it is not feasible to redo the initial safety analyses and testing every time the agent learns something. However, as we have shown in this paper, it is not always possible to just analyze the learned rule to determine its affects upon the agent's performance. We have shown that the effects of adding a new search control can be very counter-intuitive.

Therefore, we need a set of ICP conditions, C, for the planner that guarantees the ICP property for the agent's behavior with respect to its performance dimensions. The ICP property says that if the rules in current set of search control rules satisfy C and the agent is safe, then if the new search control rule also satisfies C and is safe, then the agent continues to be safe and to have this ICP property. Given such a set of ICP conditions, if we can show that the rules produced by the learner are always safe and always satisfy those conditions, then if the initial set of rules are safe and satisfy those ICP conditions then we are guaranteed that the agent continues to be safe as it learns.

We did this for a specific agent which has Prodigy as its planner and Bacall as its learner. We specified its ICP condition and showed that Bacall-learned rules were guaranteed to both satisfy that condition and to be safe.

[2]Where the node's state only describes the current world state and the current goal stack as described in 2.1.

[3]In Table 1, we see the reject rule's precondition querying the status of an edge not directly on the path.

4. REFERENCES

[1] M. Barley. *Model-Based Refinement of Search Heuristics*. PhD thesis, Department of Computer Science, Rutgers University, May 1996.

[2] M. Barley and H. Guesgen. Meta-level preconditions: An obstacle to safe SCR learning. In *IC-AI 2001 Conference Proceedings*, LasVegas, Nevada,USA, June 2001. 2001 International Conference on Artificial Intelligence, CSREA Press. To appear.

[3] A. Barrett, D. Christianson, M. Friedman, C. Kwok, K. Golden, S. Penberthy, Y. Sun, and D. Weld. Ucpop: User's manual (version 2.0). Technical Report 93-09-06d, Dept. of Computer Science and Engineering, University of Washington, 1995.

[4] J. Carbonell, J. Blythe, O. Etzioni, Y. Gil, R. Joseph, D. Kahn, C. Knoblock, S. Minton, A. Perez, S. Reilly, M. Veloso, and X. Wang. Prodigy 4.0: The manual and tutorial. Technical Report CMU-CS-92-150, Carnegie Mellon University, 1992.

[5] D. Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32(3), 1987.

[6] D. Gordon. Asimovian adaptive agents. *Journal of Artificial Intelligence Research*, 13:95–153, 2000.

[7] A. Jonsson, P. Morris, N. Muscettola, and K. Rajan. Planning in interplanetary space: Theory and practice. In *Proceedings of the Fifth International Conference on Artificial Intelligence Planning Systems*, 2000.

[8] J. Laird, C. Congdon, and K. Coulter. *The Soar User's Manual (Version 8.2)*. Computer Science Dept., University of Michigan, 23 June 1999.

[9] S. Minton. *Learning Search Control Knowledge*. Kluwer Academic Publishers, 1988.

[10] T. Mitchell, R. Keller, and S. Kedar-Cabelli. Explanation-based generalization: A unifying view. *Machine Learning*, 1(1), 1986.

[11] N. Muscettola, P. Nayak, B. Pell, and B. William. Remote agent: To boldly go where no AI system has gone before. *Artificial Intelligence*, 103(1-2):5–48.

[12] D. Weld and O. Etzioni. The first law of robotics. In *Proceedings of the Twelth National Conference on Artificial Intelligence*, pages 1042–1047, 1994.