Integrating Partial-Order Planning into the Orca Schema-Based Mission Controller[*]

Prabha Ramakrishnan
Ragula Systems, Inc.
404 East 4500 South Street
Suite A-22
Salt Lake City, Utah 84107
(801) 281-3434
slcar@am.dames.com

Roy Turner
Department of Computer Science
5752 Neville Hall
University of Maine
Orono, Maine 04469
(207) 581-3909
rmt@umcs.maine.edu

Abstract

Reasoning about plans and actions is fundamental to intelligent control of real-world systems such as AUVs. These plans can come from a library of existing plans or from creating a new plan if no appropriate plan exists for the situation. The Orca project [Turner, 1994] is an ongoing effort to create an intelligent system for mission-level control of autonomous underwater vehicles. Orca's primary means of achieving goals is to retrieve existing plans from memory and to apply them by doing their steps. This paper describes our effort to integrate partial-order planning into Orca's schema-based planning mechanism. With this work, Orca first looks for an appropriate plan in its repertoire of plans to achieve a goal or a set of goals. If such a plan is not available, it considers more general plans. For some goals, even a generalized plan may not be readily available. In such cases, Orca uses *partial-order planning* to generate a specific plan "from scratch" for the problem-solving situation. This paper describes our implementation of such a planner and its seamless integration into Orca's schema-based reasoning mechanism. Orca is now capable of creating new plans if necessary, criticizing and patching a plan (if needed) just prior to execution of the plan, and handling conjunctive, disjunctive, and negated goals.

Introduction

AUVs have great promise for use as "underwater satellites" to provide on-site data for long periods of time [Blidberg *et al.*, 1991], as replacements for remotely-operated vehicles (ROVs), and for industrial applications such as underwater prospecting, mining and construction. To fulfill this promise requires the development of intelligent controllers for AUVs.

Reasoning about actions and plans is fundamental to the development of intelligent machines that operate in the real world. A planning system may be visualized as having two components—plan creation and plan execution. Some planning systems pick an available plan that is appropriate for the situation at hand from a library of plans (e.g., PRS [Georgeff & Lansky, 1987]), while others create a new plan for a situation (e.g., NOAH [Sacerdoti, 1977]).

Orca [Turner, 1994; Turner, 1995] is an intelligent controller of autonomous underwater vehicles (AUVs) being built at the University of Maine. Prior to this work, Orca operated solely as the former type of planner, always looking for existing plans in its plan library specific to the goal and situation. If a specific plan did not exist for the given situation, Orca would use a more generalized plan.

[*] This work was supported in part by the National Science Foundation under grant number BES-9696044. The authors would like to thank the other members of the Maine Cooperative Distributed Problem Solving Research Group for helpful discussions during the work. A more complete treatment of this subject can be found in [Ramakrishnan, 1987], available on the Web as http://cdps.umcs.maine.edu/Papers/Theses/Prabha.Ramakrishnan.

The use of such pre-compiled plans has its advantages. Plan creation in the worst case is NP-hard [Chapman, 1985]. It would be computationally expensive to create plans (even the high-level plans that Orca uses) for every situation encountered by an AUV, and there are many missions for which using pre-existing plans for goals and subgoals is sufficient.

However, it is impossible to anticipate all possible situations that an AUV may encounter. A planning system in the real world needs the ability to create a plan if a pre-existing plan fails to achieve a given goal or a set of goals. It also needs to criticize and patch/repair existing plans (if needed) at run time, just prior to execution of the plan, as the world may have changed in the interval between plan creation and execution.

This paper describes our approach to enhance Orca's planning abilities by incorporating the above features. We use partial-order planning ideas from the POP (partial-order planner) program and some features of UCPOP, an extension to POP [Weld, 1994; Russell & Norvig, 1995]. Partial-order planners use a representation for plans in which the steps have a partial order, rather than committing to a total ordering during creation of the plan. Over-commitment to details of the plan, such as the order of steps and particular bindings for variables, is not desirable. It focuses attention on details at the expense of more important features of the plan. Linear, non-partial-order planners also may fail to find a plan even when one is possible, due to over-commitment (i.e., they are not *complete*). Partial-order planning uses a least-commitment strategy to avoid over-commitment to the details of a plan as it is being constructed.

We also use partial-order planning to criticize an existing plan before it is executed by the agent [cf. Sacerdoti, 1977]. Modifications to the plan are suggested to make it more fool-proof. We have also added the ability to handle conjunctive/disjunctive and negated goals to Orca.

In the remainder of this paper, we first discuss Orca and planning in general. We then describe the work done to add partial-order planning to Orca.

Orca

Orca is a schema-based, adaptive reasoner for AUV control. Adaptive reasoning is the ability of a reasoner to intelligently change its behavior, both short-term and long-term, in response to the changing needs of the problem-solving situation. Orca is an artificial intelligence (AI) program. AUV control is an ideal problem for AI research for several reasons. First, it is a real-world domain beset with incomplete knowledge and uncertainty, and it involves the control of a real physical system in a world with unpredictable processes and agents. Second, the time course of AUV actions is such that it is reasonable to expect an AI system to be fast enough. And third, research in the AUV domain can be used as a model for a large number of other real-world control problems (e.g., controlling space missions).

Orca uses knowledge structures called schemas. A *schema* is defined as an explicit, declaratively represented packet of knowledge representing either a pattern encountered (or expected) in the world or a pattern of action for the reasoner to take. The schemas are of two types. Contextual schemas (c-schemas) provide information about the current situation, whereas procedural schemas (p-schemas) provide suggestions about how to achieve goals and provide information that guides almost all of the reasoner's behavior A schema can be provided to a reasoner by another agent or, ultimately, learned from the agent's own experience.

Orca is designed to reside at the top of a layered software architecture, the EAVE AUV control architecture [Blidberg & Chappell, 1986]. This frees Orca from having to deal with low-level, time-critical issues such as sensor management and motion control. To Orca, the rest of the layers appear as a "black box" with well-defined inputs and outputs. These layers carry out commands from Orca and provide it with information about the state of the vehicle and the world.

Orca is organized into modules. Figure 11 provides an overview of the internal structure of Orca and how its modules interact with one another. Agenda Manager (AM) is responsible for focusing attention on a particular goal that Orca needs to achieve at a given time by choosing the goal with the highest importance from the agenda. Schema Applier (SA) finds, interprets, and applies procedural schemas (p-schemas) to achieve the goal selected by AM. The Communication Module is responsible for handling messages from the user and other AUVs. Event Handler (EH) manages all input from the low-level architecture and elsewhere (via the Communication Module), and is responsible for detecting and handling both anticipated and unanticipated events. Context Manager keeps other modules informed about Orca's current context and provides information to allow them to behave appropriately for it. At each point, SA tries to achieve the goal that is in focus on the agenda. To achieve this goal, it asks the long-term memory (LTM) for an appropriate p-schema based on the features of the current problem-solving situation. Orca takes actions based on the information contained in the procedural schema in a process called *schema application*.

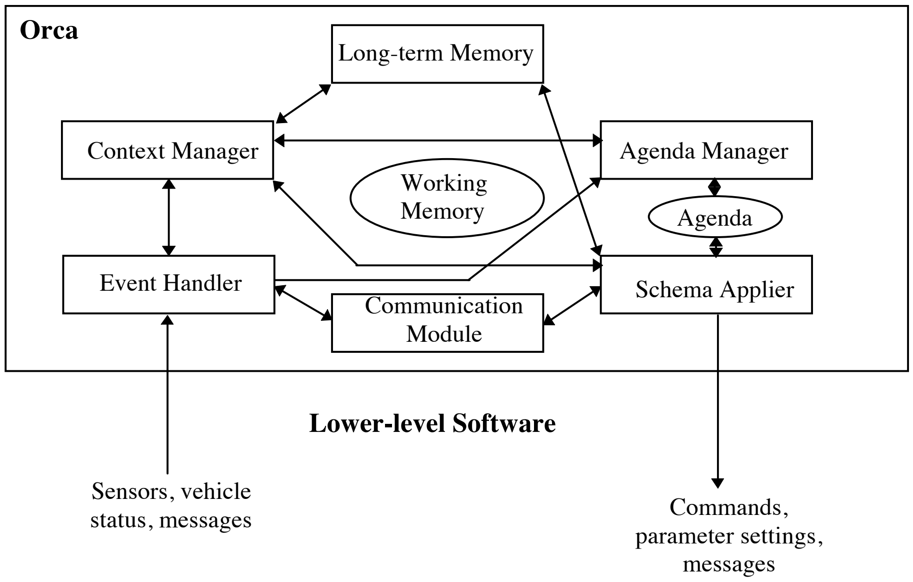

Figure 1: Internal structure of Orca.

A procedural schema (p-schema) contains a description of the situation for which it is appropriate and information about what steps to take to achieve a goal, and in what order. For example, to achieve the goal of taking a photograph of a target at (x, y, z), Orca might use a p-schema containing steps for: moving near the target; orienting to the target; and actually taking the picture. Each of these steps may be a primitive, executable action (*xact*) or a more complex action.

In general, a step can be:

- a p-schema, in which case LTM is checked for a more specialized version of the p-schema for the particular situation, then the resulting p-schema (or the original, if no specializations are found) is recursively applied;
- a goal, in which case a suitable p-schema is found for it and applied; or
- an executable action (xact) that Orca carries out directly or by issuing a command to lower-level software.

A schema-based reasoner's primary memory structures are its schemas. Appropriate schemas are retrieved from memory depending on the problem-solving situation. For a procedural-schema, the important features are the goal to be achieved and the attributes of the current situation related to the schema's current needs and application conditions. Orca uses a conceptual memory modeled after the CYRUS program [Kolodner, 1984; see Turner, 1994] to store and retrieve its p-schemas. This mechanism for selecting p-schemas means that Orca finds the most specific p-schema for the goal and situation by traversing a set of specialization hierarchies in which more general p-schemas index more specific ones by features differentiating between them.

Orca can only achieve goals for which a p-schema exists in memory. However, this is not the limitation it may appear. The idea is that Orca's memory will contain schemas which themselves carry out the creation of new p-schemas. This facility is part of what this work added to Orca.

Planning

A problem of considerable interest in the field of automated reasoning has been the design of systems which can plan, that is, systems that can describe a set of actions that will later allow them to achieve a desired goal or mission. Such systems are referred to as *planners*. An AI planning system is charged with generating a *plan* which is one possible *solution* to a specified *problem*. A *problem* is characterized by an initial state description and a final state description.

The plan generated will be composed of operators. A typical operator consists of three components:

a. **action description** - a possible action along with its parameters

b. **preconditions** - a conjunction of facts that must hold before the operator can be applied; i.e., a partial description of a state in which the operator can be used

c. **effects** - a conjunction of facts that are expected to hold after the operator has been applied

For example, consider an action to take a sample from a location *loc*:

Action: take-sample(loc)
Preconditions : (location AUV loc)
Effects: (have-sample AUV loc)

The goal of a planner is to create a plan, which is an ordered set of operators, that when applied to some initial state will give rise to the desired final state. A complete plan is one in which every precondition of every step (operator) is achieved by the initial state or some other operator in the plan. A consistent plan is one in which the ordering of the steps is consistent and the bindings of any variables are also consistent.

A common way of representing a plan is as a data structure consisting of four components [e.g., Weld, 1994]:

- the set of *steps* used in the plan;
- a set of *ordering constraints*, each of which specifies the relative ordering of two steps in the plan; these are often written as $S_i < S_j$, meaning that S_i precedes S_j in the plan;
- a set of *causal links,* or *protected links*, that specify conditions that must be protected between the end of one step and the beginning of another; these are often represented as triples (S_i, S_j, c), which means condition *c* is to be protected from the end of S_i to the beginning of S_j; and
- a set of *variable binding constraints* linking variables to objects in the world or to other variables.

There are two major categories of planners: total-order (or linear) and partial-order (or nonlinear) planners. Linear planners commit to one of the goals (or preconditions) in the problem and work on that until it is solved, then work on the next, and so on. They typically plan by searching a space of states, each of which is one configuration of the problem, for a solution. Nonlinear planners do not commit to solving a particular goal before moving on to others, but instead may work on several goals at once. They typically operate by searching a space of partial plans rather than problem configurations. This allows a beneficial level of abstraction in their operation: they use abstract operations on plans (e.g., add a step, add an ordering constraint, etc.) that are domain-independent. These planners start with a partial plan having no steps or ordering constraints and move toward a final state that consists of a complete plan that achieves the goals(s). The planning mechanism added to Orca is based on the ones used by nonlinear, partial-order planners.

Partial-Order Planning in Orca

This work added a partial-order planning module, called POP[1], to Orca. Rather than have Orca's Schema Applier watch for goals it cannot handle and invoke this module directly, however, we take an approach that integrates the use of POP into Orca's existing schema application mechanism. In this approach, Orca's knowledge, not its control structure, determines when POP is invoked to create a new plan.

When Orca encounters any goal to be achieved, it looks in its schema memory for the most specific p-schema possible for that goal. In the case of novel goals for which there are no specific p-schemas, the memory will return the starting point of its search, the top-level p-schema *p-achieveGoal*. The idea is that this schema should have steps that achieve the "meta-goal" of achieving the goal. It is in this schema that we put an xact that invokes POP to create a new p-schema. *P-achieveGoal* also contains a step that tells SA to execute the new p-schema. To Orca, then, a novel goal is just like any other goal: the best p-schema is found for it and applied. The only difference is that in this case, the schema applied causes a new schema to be created.

There are several advantages to this approach, chief of which is the ability to add several different ways to create p-schemas to Orca, each of which is best used in different situations. This would be done by creating p-schemas for the various methods and indexing those in Orca's memory according to the situations in which they will be useful. By making explicit the invocation of a p-schema creation mechanism, rather than embedding it in Orca's structure, we also leave open the possibility that Orca can learn from its experience which methods are best for which situations.

[1] In honor of but the module should not be confused with Weld's [1994] POP and UCPOP programs.

The POP module is also used to criticize existing p-schemas just prior to their application. This is handled via an explicit invocation from Schema Applier.

In this section, we discuss the POP module and how it is used to create new p-schemas. We then discuss how it is used for p-schema criticism.

The Partial-Order Planning Module

Plan representation

A plan contains information about its steps, the ordering of those steps, the links (causal information) between them, and the bindings for the variables appearing in the preconditions and effects of the steps. A plan in the partial-order planning (POP) module is an association list representing a four-tuple containing this information, for example:

```
((steps (s1 s2, ...))
 (order ((< s1 s2) (< s2 s3)...))
 (links ((s1 s2 c) ...))
 (bindings ((x 100) (y 200)...)))
```

Our implementation of POP supports plan steps that are either p-schemas or xacts. As an example of an xact, consider *^move-to*, which is used to issue a command to the AUV to move to a location:

```
(defframe ^move-to (^issue-command)
      (parameters (location))
      (location)
      (command (move-to @location))
      (preconditions ((location $self (?ix ?iy ?iz))
                     (not(location $self (?x ?y ?z)))))
      (effects ((location $self (?x ?y ?z))
               (not(location $self (?ix ?iy ?iz)))))
      (parms-from-effs ((?x ?y ?z)))
      (effs-from-parms ((location $self @location)
                       (not(location $self (?ix ?iy ?iz)))))...)
```

In our implementation, each xact is a frame [see, e.g., Minsky, 1975]. Each frame contains slot/value pairs. In the above example, parameters, location, command, etc., are the slots. The notation "@var" is used to refer to a slot value in the same frame, and "?var" is a variable.

An xact's *parameters* slot contains the names of its parameters. The value of each is stored in a slot with the same name as the parameter. The *preconditions* slot and the *effects* slot contain descriptions of conditions that must be true prior to the xact's execution and those that are expected to be true after its execution, respectively. In addition to these slots, POP also needs information about how to translate between its internal plan representation and Orca's p-schema representation. This information is contained in the slots *parms-from-effs* and *effs-from-parms*. (See Ramakrishnan [1997] for details about these slots' use.)

Planning

The POP algorithm is shown in Figure 22. This is a non-deterministic algorithm that takes as input an initial partial plan, an empty agenda of things to do, and a list of known operators, and that returns a complete plan. The initial plan contains a specification of the initial state and the goal state. It contains two steps, *^start* and *^finish*:

```
Steps:
    ^start:                                ^finish:
        preconditions: nil                     preconditions: <goal conditions>
        effects: <initial conditions>          effects: nil
Order:  ^start < ^finish
Links: nil
Bindings: nil
```

The start step's "effects" assert the initial conditions, while the goals are taken to be the preconditions of the finish step. Since POP is embedded in a larger reasoner in our work, we have made a slight change from this scheme in that the contents of Orca's working memory at planning time is taken as the initial conditions. In addition, the POP algorithm is wrapped in functionality for Orca that takes the final plan and converts it into Orca's p-schema representation.

```
function POPALGORITHM(plan, pop-agenda, operators) returns plan
  loop do
    if SOLUTION?(plan) then return plan
    $S_{need}$, c = SELECT-SUBGOAL(plan)
    CHOOSE-OPERATOR(plan, operators, $S_{need}$, c)
    RESOLVE-THREATS(plan)
  end
end

function SELECT-SUBGOAL(plan) returns $S_{need}$, c
   pick a plan step $S_{need}$ from STEPS(plan) with a  precondition 'c' that has not been achieved
   return $S_{need}$, c
end

function CHOOSE-OPERATOR(plan, operators, $S_{need}$, c)
 choose a step $S_{add}$ from operators or
    STEPS(plan)that has '$c_{add}$' as an effect such that u = UNIFY(c, $c_{add}$, bindings(plan))
 if there is no such step then fail
   add 'u' to bindings(plan)
   add ($S_{add}$, $S_{need}$, c) to LINKS(plan)
   add (< $S_{add}$ $S_{need}$) to ORDERINGS(plan)
   if $S_{add}$ is a newly instantiated step from operators  then
      add $S_{add}$ to STEPS(plan)
      add (Start < $S_{add}$ < Finish) to ORDERINGS(plan)
end

function RESOLVE-THREATS(plan)
 for each link ($S_i$, $S_j$, c) in LINKS(plan) do
  for each $S_{threat}$ in STEPS(plan) do
   for each $c_{new}$ in EFFECTS($S_{threat}$) do
   ;; Remove redundant nots
   c = REDUCE-NEGATIONS(c)
   $c_{new}$ = REDUCE-NEGATIONS ($c_{new}$)
   if UNIFY(c,   $c_{new}$, bindings(plan)) then
      choose either
         Promotion : Add (< $S_{threat}$ $S_i$) to ORDER
         Demotion : Add (< S(j) $S_{threat}$) to ORDER
         if not CONSISTENT(plan) then fail
   end
  end
 end
end
```

Figure 2: The POP algorithm. (Modified after [Weld, 1994; Russell & Norvig, 1995].)

POP starts with the initial partial plan. On each cycle, the plan is expanded by achieving a precondition *c* of a step S_{need}. The function *select-subgoal* returns this step and the condition to work on. An operator is selected that can achieve *c*. This operator may be one already in the plan, one from the list of xacts, or a p-schema found in Orca's memory by presenting the memory with *c* as a memory probe. *Choose-operator* also records the causal link for the newly-achieved precondition.

Next, the plan is examined for the existence of threats. A *threat* is detected whenever a plan step potentially violates a causal link occurring in the plan. For example, suppose a plan contains two steps, move to location *A* (*S1*) and take a picture of the object at *A* (*S2*), with a causal link *c* ("preserve at *A* between *S1* and *S2*") between them to denote that the vehicle should not move from *A* until after *S2* is complete. If a new step, move to location *B* (*S3*), is added to the plan, one of its effects ("at *B*") conflicts with *c*. Threats are handled using one of two threat-resolution strategies in our version of POP. *Promotion* moves the threatening step after the last step in the link, and *demotion* moves the threatening step before the first step in the link.

The POP algorithm uses the non-deterministic primitives *choose* and *fail*. Orca is implemented in Lisp, which does not support non-determinism directly. Consequently, these primitives are implemented using recursion and a list of alternative choices. When *choose* is encountered, the choice point is remembered along with other possible choices. If a *fail* is encountered, control returns to this point and another choice is made. This occurs until one of the choices succeeds or there are no more choices, in which case a failure occurs at the *choose* operator. For a recursive, rather than non-deterministic, version of this algorithm, see Ramakrishnan [1997].

Integration into Orca

As discussed previously, POP is called to create a new plan from an executable action (xact) inside the p-schema *p-achieveGoal*, which is shown in Figure 33. When Schema Applier needs to achieve a goal *G* that Orca has not encountered before, then the memory automatically returns this p-schema. SA instantiates the schema by creating an instance of it containing *G* in the "goal" slot, then begins application of the schema. The first step, *call-pop*, invokes POP to create a plan for the goal. POP returns this plan as a p-schema in the special variable *$result*. The description of the next step in the schema tells SA to apply the p-schema just created. (*Do-action* is a "control step", which does not reside in a frame slot.)

```
(defframe P-achieveGoal (^p-schema)
        (actor $self)
        (parameters)
        (goal (^achievement-goal))
        (steps (call-pop))
         (call-pop (action (^pop-command (parameters ((goal @goal))))))
        ;;; Pop-command is an xact, whose xact
        ;;; function, xfcn-pop
        ;;; calls POPALGORITHM
        (order (sequential call-pop
                        (do-action (action ($result))
                                (parameters (goal @goal)))))
        ;;; $result: holds the value that is
        ;;; returned by POPALGORITHM - new
        ;;; p-schema
        )
```

Figure 3: P-achieveGoal.

The POP algorithm was originally meant to be a standalone program [e.g., Weld, 1994]. To integrate it into an existing planning system, several changes and additions were necessary [see Ramakrishnan, 1997]. We implemented POP as a CLOS (Common Lisp Object System) class. Orca contains an instance of POP as one of its modules; the interface between POP and the rest of Orca is via POP's methods. Internally, POP works as described above, using an internal representation very similar to the original representation in [Weld, 1994]. Orca's representation of plans and goals is somewhat different. Consequently, when POP is invoked, conversion must occur between Orca's goal representation and POP's, and then from POP's plan representation back to Orca's p-schema representation at the end of planning.

Figure 44 shows a plan created by POP as a result of Orca being given the conjunctive goal: take a sample from location (200,200,20), take a photograph of location (150,150,10), and move to another location (160,160,10). Figure 55 shows the p-schema resulting from POP's plan. The plan is: move to (200,200,20), take the sample, move to (150,150,10), take the photograph, move to (160,160,10).

```
((STEPS
  ((^MOVE-TO6) (^MOVE-TO3) (^TAKE-PHOTOGRAPH0) (^ORIENT-TO0) (^MOVE-TO0)
   (^TAKE-SAMPLE0) (^STARTP) (^FINISH)))
 (ORDER
  ((< (^TAKE-PHOTOGRAPH0) (^MOVE-TO6)) (< (^MOVE-TO3) (^MOVE-TO6))
   (< (^STARTP) (^MOVE-TO6)) (< (^MOVE-TO6) (^FINISH))
   (< (^TAKE-SAMPLE0) (^MOVE-TO3)) (< (^MOVE-TO0) (^MOVE-TO3))
   (< (^MOVE-TO3) (^FINISH)) (< (^STARTP) (^MOVE-TO3))
   (< (^MOVE-TO3) (^TAKE-PHOTOGRAPH0)) (< (^STARTP) (^TAKE-PHOTOGRAPH0))
   (< (^TAKE-PHOTOGRAPH0) (^FINISH)) (< (^STARTP) (^ORIENT-TO0))
   (< (^ORIENT-TO0) (^FINISH)) (< (^MOVE-TO0) (^FINISH))
   (< (^STARTP) (^MOVE-TO0)) (< (^MOVE-TO0) (^TAKE-SAMPLE0))
   (< (^STARTP) (^TAKE-SAMPLE0)) (< (^TAKE-SAMPLE0) (^FINISH))
   (< (^STARTP) (^FINISH))))
 (LINKS
  (((^MOVE-TO3) (^MOVE-TO6) (LOCATION $SELF (?IX16 ?IY16 ?IZ16)))
   ((^STARTP) (^MOVE-TO6) (NOT (LOCATION $SELF (?X25 ?Y25 ?Z25))))
   ((^MOVE-TO6) (^FINISH) (LOCATION $SELF (160 160 10)))
   ((^MOVE-TO0) (^MOVE-TO3) (LOCATION $SELF (?IX10 ?IY10 ?IZ10)))
   ((^STARTP) (^MOVE-TO3) (NOT (LOCATION $SELF (?X15 ?Y15 ?Z15))))
   ((^MOVE-TO3) (^TAKE-PHOTOGRAPH0) (LOCATION $SELF (?PX7 ?PY7 ?PZ7)))
   ((^TAKE-PHOTOGRAPH0) (^FINISH) (PHOTOGRAPH $SELF (150 150 10)))
   ((^STARTP) (^ORIENT-TO0) (NOT (HEADING $SELF ?HEADING5)))
   ((^ORIENT-TO0) (^FINISH) (HEADING $SELF 66))
   ((^STARTP) (^MOVE-TO0) (LOCATION $SELF (?IX1 ?IY1 ?IZ1)))
   ((^STARTP) (^MOVE-TO0) (NOT (LOCATION $SELF (?X2 ?Y2 ?Z2))))
   ((^MOVE-TO0) (^TAKE-SAMPLE0) (LOCATION $SELF (?SX1 ?SY1 ?SZ1)))
   ((^TAKE-SAMPLE0) (^FINISH) (SAMPLE $SELF (200 200 20)))))
 (BINDINGS
  ((SX ?SX1) (SY ?SY1) (SZ ?SZ1) (SX1 200) (SY1 200) (SZ1 20) (IX ?IX1)
   (IY ?IY1) (IZ ?IZ1) (X ?X2) (Y ?Y2) (Z ?Z2) (X2 200) (Y2 200) (Z2 20)
   (NIL NIL) (IX1 120.0d0) (IY1 120.0d0) (IZ1 0.0d0) (HEADING ?HEADING5)
   (HEADING5 66) (PX ?PX7) (PY ?PY7) (PZ ?PZ7) (PX7 150) (PY7 150) (PZ7 10)
   (X15 150) (Y15 150) (Z15 10) (IX10 ?X2) (IY10 ?Y2) (IZ10 ?Z2) (X25 160)
   (Y25 160) (Z25 10) (IX16 ?X15) (IY16 ?Y15) (IZ16 ?Z15))))
```

Figure 4: A plan created by POP.

```
^PROCEDURAL-SCHEMA0 is a frame with the following description:
 NAME:              PROCEDURAL-SCHEMA0
 ISA:               (^PROCEDURAL-SCHEMA)
 SLOTS:
   o ACTOR: $SELF
   o BINDINGS: [….] ; Same as above plan
   o CONCEPT-OWNER: -none-
   o EFFECTS: ((SAMPLE $SELF (200 200 20)) (HEADING $SELF 66) (PHOTOGRAPH $SELF
              (150 150 10)) (LOCATION $SELF (160 160 10)))
   o FINISH: (STEP (ACTION (^FINISH (PARAMETERS NIL))))
   o LINKS: [….] ; same as above plan
   o MOVE-TO0: (STEP (ACTION (^MOVE-TO (PARAMETERS ((LOCATION (200 200 20)))))))
   o MOVE-TO3: (STEP (ACTION (^MOVE-TO (PARAMETERS ((LOCATION (150 150 10)))))))
   o MOVE-TO6: (STEP (ACTION (^MOVE-TO (PARAMETERS ((LOCATION (160 160 10)))))))
   o ORDER: (SEQUENTIAL STARTP MOVE-TO0 TAKE-SAMPLE0 MOVE-TO3
             TAKE-PHOTOGRAPH0 MOVE-TO6 ORIENT-TO0 FINISH)
   o ORIENT-TO0: (STEP (ACTION (^ORIENT-TO (PARAMETERS ((HEADING 66))))))
```

Figure 6 A p-schema representing the POP plan in Figure 4.5 (Continued on next page)

```
o PLANSTEPS: ((^MOVE-TO6) (^MOVE-TO3) (^TAKE-PHOTOGRAPH0)
                   (^ORIENT-TO0) (^MOVE-TO0) (^TAKE-SAMPLE0) (^STARTP) (^FINISH))
o RAW-PLAN: [....] ; same as the POP plan above
o STARTP: (STEP (ACTION (^STARTP (PARAMETERS NIL))))
o STEPS: (STARTP MOVE-TO0 TAKE-SAMPLE0 MOVE-TO3 TAKE-PHOTOGRAPH0
             MOVE-TO6 ORIENT-TO0 FINISH)
o TAKE-PHOTOGRAPH0: (STEP (ACTION (^TAKE-PHOTOGRAPH
                                     (PARAMETERS ((LOCATION (150 150 10)))))))
o TAKE-SAMPLE0: (STEP (ACTION (^TAKE-SAMPLE (PARAMETERS ((LOCATION
                                      (200 200 20)))))))
```

Figure 5: P-schema version of previous plan.

The addition of the POP module to Orca also allows the program to achieve more complex goals than it could before. This includes conjunctive, disjunctive, and negated goals. For example, when a conjunctive goal such as the one just mentioned is encountered, Orca may have p-schemas that can achieve each of the conjuncts, but lack a p-schema that can achieve the conjunction. In this case, POP is automatically invoked, since this is, in effect, a novel goal. POP will create a new p-schema using the old p-schemas as steps and "gluing" them together as necessary.

Procedural Schema Criticism

It is not sufficient to simply retrieve stored plans an apply them. The current situation may be different from the situation in which the plan was created. This is true even of plans created by p-achieveGoal during the same mission, because Orca can interrupt execution of a plan as the situation demands; consequently, the situation may change between the time the plan is created and when it is finally applied. In addition, it is often not possible to create a complete plan prior to the time of execution, since information about the state of the world may not be available at the start of a mission.

We address these problems by criticizing p-schemas just prior to execution. Criticism entails identifying threats and incomplete plans. Incomplete plans are those that have preconditions that are not achieved by the current situation or steps already existing in the p-schema. Once a problem has been detected, the p-schema is *patched* by adding steps or reordering steps, as necessary. At that point, schema application can continue.

Criticism of p-schemas is done by the POP module. Schema Applier calls POP just prior to applying a p-schema. POP translates the p-schema into its plan representation, recovering as much causal link information from the p-schema as possible. The resulting plan is a partial plan, and criticism from this point on is identical to using the POP algorithm to create a new plan, given the partial plan. When POP is finished, it converts the criticized plan back to Orca's p-schema representation and application proceeds.

Criticism is currently not done on plans that lack causal information. This includes conventional plans that have no causality and those created by hand, when the user specified no causal information. Conventional plans are those for which there are no causal reasons for doing particular steps in a particular order, or if there are, the planner does not have access to that information. Many social plans (e.g., going to the movies) have or contain subplans that have a conventional nature. We anticipate there being such plans in the AUV domain as well.

Conclusion and Future Work

Though a schema-based reasoner such as Orca or PRS has many benefits (especially efficiency), if they can only use existing plans, they are handicapped when compared to a from-scratch planner. By adding the ability to create new schemas to such a reasoner, one gains the benefits of both kinds of reasoners.

In this paper, we have described how from-scratch planning was added to the Orca schema-based reasoner. By encoding the planning mechanism as an executable action that is part of a schema, Orca now uses existing plans or creates new ones as the situation demands, without "conscious" effort on its part. This approach also has benefits for the future, since it opens the door Orca having multiple ways of creating plans (e.g., partial-order planning, constraint satisfaction, case-based reasoning, etc.) from which it can choose at run-time as the situation calls for. It also opens the door to Orca being able to learn from its own experience when to use from-scratch planning, and what type.

Much work remains to be done. The partial-order planning algorithm we used is very simple, and cannot handle all that we might like it to. For example, it cannot handle quantified goals, nor can it do temporal or resource-based reasoning. For some of these problems, algorithms are readily available, such as UCPOP [Weld, 1994] for quantified goals and DEVISER [Vere, 1983] for temporal planning. Future work on Orca will examine integrating these algorithms into the reasoner. Another area of future work has to do with identifying those situations in which one kind of from-scratch planning is better than another. Another important area is determining in a more principled way when to use from-scratch planning. At the moment, we use it whenever no p-schema other than *p-achieveGoal* is available. However, there are likely situations in which a more specific p-schema *could* be used, but should not be used. And finally, work is needed to identify how to store the results of from-scratch planning, the schemas as well as the results of having applied them.

References

Blidberg, D.R., and Chappell, S.G. (1986). Guidance and control architecture for the EAVE vehicle. *IEEE Journal of Oceanic Engineering*, v. OE-11, no. 4, pp. 449-461.

Blidberg, D.R., Turner, R.M., and Chappell, S.G. (1991). Autonomous underwater vehicles: Current activities and research opportunities. *Robotics and Autonomous Systems*, v. 7, pp. 139-150.

Chapman, D. (1985). Planning for conjunctive goals. *Artificial Intelligence*, v. 32 no. 3, pp. 333-378.

Fikes, R. E. & Nilsson, N. J. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence,* 2:189-208.

Georgeff, M. P., and Lansky, A.L. (1987). Reactive reasoning and planning: An experiment with a mobile robot. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, pp. 677--682, Seattle, Washington.

Minsky, M.L. (1975). A framework for representing knowledge, in *The Psychology of Computer Vision*, P. H. Winston (ed.), McGraw-Hill, New York.

Ramakrishnan, P. (1997). *Intelligent Control of Autonomous Underwater Vehicles - A Partial-Order Planner for the Orca Project.* Master's thesis, Department of Computer Science, University of Maine, Orono, Maine. Available on the Web as: http://cdps.umcs.maine.edu/Papers/Theses/Prabha.Ramakrishnan.

Russell, S. & Norvig, P. (1995). *AI - A modern approach.* Prentice Hall, 1995.

Sacerdoti, E. D. (1975). The nonlinear nature of plans. *IJCAI,* pages 206-214.

Turner, R. M. (1994). *Adaptive Reasoning for Real-World Problems: A Schema-Based Approach.* Lawrence Erlbaum Associates, Publishers, Hillsdale, New Jersey.

Turner, R. M. (1995). Context-sensitive, adaptive reasoning for intelligent AUV control: Orca project update. In *Proceedings of the Ninth International Symposium on Unmanned, Untethered Submersible Technology* (UUST '95), Durham, New Hampshire.

Vere, S.A. (1983). Planning in time: Windows and durations for activities and goals. *Pattern analysis and Machine Intelligence (IEEE)*, v. 5, pp. 246-67.

Weld, D.S. (1994). An introduction to least commitment planning. *AI Magazine,* Winter 1994, pages 27 - 61.