Knowledge Acquisition Tools for Planning Systems

Marie desJardins
SRI International
333 Ravenswood Ave.
Menlo Park CA 94025
marie@erg.sri.com

Abstract

Knowledge engineering is a critical task in the development of AI planning applications. In order to build large-scale, real-world planning applications, tools must be developed that will provide efficient, effective ways to create, modify, debug, and extend the knowledge bases for such systems. As much as possible, this development and updating process should be automated.

The goal of this project is to develop knowledge acquisition tools for AI planning systems. We have developed a graphical Operator Editor, which allows users to develop new planning operators and revise existing operators, and the Operator Learner, an inductive learning-based tool for knowledge engineering. Both tools have been implemented within the SOCAP planning system.

This paper describes these two tools, discusses related work, and summarizes directions for future research.

Introduction

Knowledge engineering is a critical task in the development of AI planning applications. For such systems, a significant part of the development effort goes into writing and debugging planning operators. This process typically requires an AI expert, making knowledge engineering costly, tedious, and prone to errors.

The goal of our research is to develop knowledge acquisition tools that can simplify the knowledge development and debugging process, and that will be usable by non-AI experts. Interactive tools can aid the user in entering planning knowledge; automated tools can verify the interactively developed knowledge and learn new knowledge from online sources. These tools should support both initial knowledge base development and knowledge acquisition "on the fly" (i.e., during planning), so that knowledge bases can be adapted and improved as the system is used. Tools that enable the system to learn, that is, to improve its performance over time, will result in systems that can be used for unforeseen types of operations and situations.

We are building two knowledge acquisition tools for SOCAP: the Operator Editor and the Operator Learner. The Operator Editor provides a graphical interface to build and modify SOCAP planning operators. The Operator Learner is an inductive learning tool that identifies appropriate preconditions for the planning operators by generalizing from feedback from evaluation models and from the user's planning choices.

We have implemented and applied both the Operator Editor and the Operator Learner in an oilspill planning domain. The Spill Response Configuration System (SRCS) is an application of SOCAP that responds to coastal oil spills and identifies equipment shortfalls (Desimone & Agosta 1993). The SRCS incorporates a spreadsheet-based evaluation model that provides the feedback required by the Operator Learner.

We briefly describe the Operator Editor, then present the Operator Learner in more detail. We then summarize related work in the field, identify directions for future work, and give our conclusions.

Operator Editor

The operator editor provides a graphical interface for developing and modifying SOCAP planning operators. In addition to its use during knowledge development, the operator editor can also be used during planning and replanning. For example, if SOCAP fails to solve a goal because no operator has a purpose that matches the goal, the user can enter the operator editor, build an operator that represents a subplan for solving that goal, and return to SOCAP; the new operator will be used to expand the goal in the plan without the need for backtracking.

The operator editor provides consistency checking and type checking throughout operator development. "Fill-in-the-blank" templates are provided for editing objects, so that the possibility of syntactic errors is avoided. A data dictionary specifying the correct format for classes, variables, and predicates is used to constrain the values entered, ensuring that these objects are of the correct type. The goal is to support users by giving them as much assistance as possible, without constraining them to a particular model of operator development.

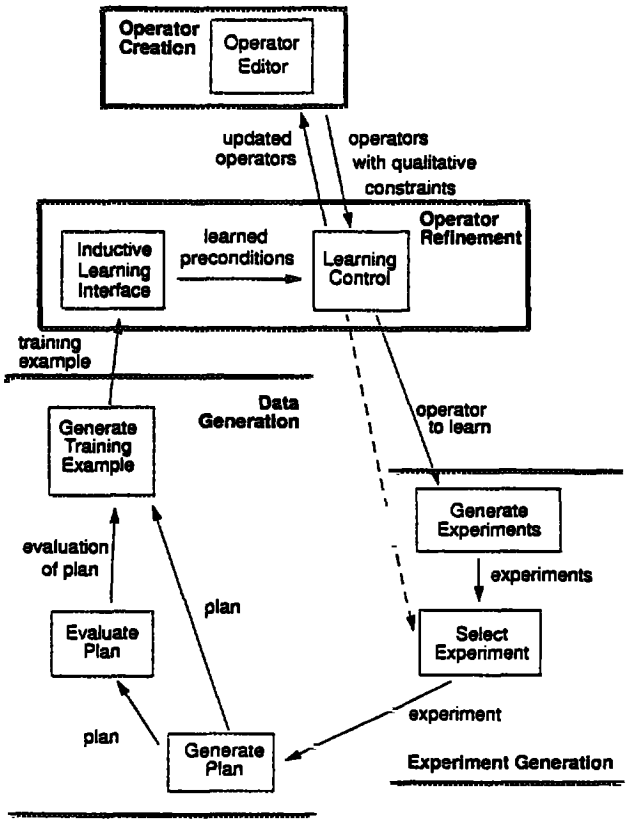

Figure 1: Architecture of the Operator Learner

Operator Learner Architecture

The Operator Learner refines the operators created by the user in the Operator Editor by learning preconditions on their applicability. The Operator Learner uses an operator's qualitative constraints (QCs)–abstract, partial knowledge specified by the user–to generate a series of experiments, each of which specifies a set of constraints on the planning process. By varying these constraints, the Operator Learner tests the quality of the plans produced by using the operator under a range of conditions. An external module evaluates the plan, and the Operator Learner again uses the QCs to extract training instances describing the world state, plan, and evaluation result.

An inductive learning module analyzes these training instances to create a hypothesis of the conditions under which the operator is expected to succeed. The result is interpreted by the Operator Learner as a precondition for the operator, and is incorporated into the operator definition, subject to the expert user's approval.

The architecture of the Operator Learner, shown in Figure 1, consists of four major components: Operator Creation, Operator Refinement, Experiment Generation, and Data Generation. These components are described in the following subsections.

Operator Creation

First, the user uses the Operator Editor to create one or more operators with partial preconditions represented as qualitative constraints. QCs represent user-provided guidance that may be partial or incomplete. They are used to guide the learning process. The intuition behind QCs is that users can often specify abstract constraints on the operator (e.g., by specifying relevant properties for determining success), even when they cannot precisely and completely specify the actual constraints. Some examples of QCs are

- "The equipment required to clean up an oil spill depends on the type and amount of oil, weather, water currents, and response time."
- "Bad weather usually delays transportation actions, and air movements are more likely to be delayed than sea movements."
- "Republicans are less likely than Democrats to vote for social programs."

None of these are precise enough to be used as preconditions by the planning system, but they significantly constrain the space of possible preconditions, making automated learning of preconditions via refinement of the QCs computationally feasible.

In the current implementation, QCs are used only to represent generalizations of preconditions: specifically, predicates with underspecified arguments. Each QC matches SIPE-2 predicates or properties of objects in the SIPE-2 sort hierarchy. Arguments to a QC can be a wild card (*) that matches anything, arguments (variables) of the operator that contains the QC, or objects or classes (the latter match any object of that class). Negated predicates are allowed.

Figure 2 shows a simplified version of the *ml-get-boom1* operator, which generates a subplan to deploy a specified amount of boom to a sea sector affected by a spill. The operator sets up a subgoal to move a selected boom to the sector, deploys the boom to the sector, and establishes an additional subgoal to deploy any additional boom that is needed. The QCs of the operator are *(%property max-sea-state boom.1 *)* and *(sea-state sea-sector.1 latest.1 *)*. The first QC refers to the *max-sea-state* property of the variable *boom.1* (i.e., the most severe level of ocean conditions under which the boom is effective). The user has indicated that the *max-sea-state* can take on any value with the * argument, so the Operator Learner will have to use its learning methods to identify the correct value(s). The second QC indicates that the *sea-state* predicate (ocean conditions) for *sea-sector.1* (the location of the operation) at time *latest.1* is relevant. Again, the * indicates that the value has not been constrained, so the system must learn the correct value.

```
OPERATOR: ml-get-boom1
PURPOSE: (level>= vessel1 boom-level1 num1);
PRECONDITION: (in-service boom1);
PROPERTIES:
  NONLOCAL-VARS = (sea-state.1 sea-sector.1
    latest.1),
  QC = ((%property max-sea-state boom.1 *)
    (sea-state.1 sea-sector.1 latest.1 *));
PLOT:
PARALLEL
  BRANCH 1:
    GOAL
      GOALS: (located boom1 sea-sector1);
    PROCESS
      ACTION: deploy-boom;
      EFFECTS: (boom-deployed boom1 vessel1),
        (produce vessel1 boom-level1 num2);
  BRANCH 2:
    GOAL
      GOALS: (level>= vessel1 boom-level1 num1);
END PARALLEL
END PLOT END OPERATOR
```

Figure 2: ML-Get-Boom1 Operator

Operator Refinement

The Operator Refinement process provides the overall control for the Operator Learner. The Learning Control subprocess decides which operator to learn (currently, this is done when the user explicitly invokes the learning system) and calls the Experiment Generation process. The Inductive Learning subprocess provides a generic interface to an external inductive learning module.

An important requirement for the inductive learning system is that it must be able to learn probabilistic concepts. Probabilistic concepts appear for various reasons: there may be uncertainty in the (simulated) world; SOCAP's representation of the world state may be incomplete; there may be incorrect beliefs in SOCAP about the state of the world at a particular point; or the correct set of preconditions may be too complex to represent or learn in a reasonable amount of time and memory, so a probabilistic summary will be required.

PAGODA is a model for an intelligent autonomous agent that learns and plans in complex, nondeterministic domains (desJardins 1992). The guiding principles behind PAGODA include probabilistic representation of knowledge, Bayesian evaluation techniques, and limited rationality as a normative behavioral goal. Although we are currently using PAGODA for the Inductive Learning subprocess, the interface has been developed in a general, transparent way that will also enable us to test the system with other public-domain learning software, such as C4.5 (Quinlan 1993).

PAGODA's inductive hypotheses are represented as sets of conditional probabilities that specify the distribution of a predicted feature's value, given a set of input features. Since SIPE-2 (and hence SOCAP) cannot represent the probabilistic theories learned by PAGODA as preconditions, we use thresholding to create deterministic preconditions. Information is lost in this process: in general, the deterministic preconditions are overly strict (i.e., they sometimes rule out an operator in a case where it is, in fact, applicable). Each rule that PAGODA learns states that in situation S, an action or operator A succeeds with probability P. The learning system analyzes the theories (rule sets) to identify situations S such that if S is true, A succeeds with probability greater than some threshold $P_{Success}$; if S is false, A fails with probability greater than another threshold $P_{failure}$. These situations are the discrimination conditions for A, and are added to the system as preconditions after they are confirmed by the user.

To give a very simple example of how inductive learning works, suppose that in *ml-get-boom1*, the result of evaluating the plan is such that regardless of the specific sea-sector and time, whenever the value of the *sea-state* predicate is 3 or less, the operator succeeds, and whenever it is 4 or more, the operator fails. The learning system would form the hypothesis

```
(sea-state sea-sector.1 latest.1 [1-3])
    => success
(sea-state sea-sector.1 latest.1 [4-5])
    => failure
```

If there is noise or randomness (e.g., in some cases the operator fails even though sea-state is 3 or less, or sometimes succeeds when sea-state is 4 or 5), the probabilistic hypothesis evaluation model built into PAGODA will determine the most probable hypothesis.

After inductive learning takes place, the learning system must decide whether to continue (i.e., generate another experiment to run) or to stop learning (when an acceptable precondition(s) based on the QCs has been learned, or when none could be found). In the initial implementation, this is done manually, by asking the user whether to continue. We are exploring various alternative methods for making this decision.

Experiment Generation

Using the QCs in the operator(s) to be learned, the Generate Experiments subprocess generates a list of experiments. Each experiment represents a set of constraints to be applied during the planning process, and consists of a problem to solve, operators to select, variable bindings to apply during operator expansion, additional world predicates to establish, and new objects and properties to define. The intuition behind the implementation of this process is that arguments to the QC that are filled with variables are either predefined (for nonlocal variables that are bound before this operator is even applied) or to be selected by the QC (for local variables whose value needs to be set based on the constraints specified in the precondition). The two types of variables require different constraints on the

planning process to establish their values. The nonlocal variables are indicated explicitly in the operator by the user; as shown in Figure 2, the nonlocal variables for *ml-get-boom1* are *sea-state.1*, *sea-sector.1*, and *latest.1*. The other arguments (* or class names) are values that provide constraints for determining whether or not to select this operator, and for instantiating the other variables within the operator. These values, however, cannot be set directly; they are dependent on the variable choices (e.g., in *ml-get-boom1*, the value of the *max-sea-state* property depends on the choice of *boom.1*).

For example, in the *ml-get-boom1* operator in Figure 2, the first QC indicates that the resource *boom.1* should be identified, at least in part, based on its *max-sea-state* value. (That is, the variable *boom.1* should be bound to a boom with an appropriate value for *max-sea-state*, where the meaning of appropriate is what must be learned in order to refine the precondition.) Moreover, selecting a boom with an inappropriate value for *max-sea-state* may cause the operator to fail.

The second QC indicates that the sea-state at the place and time of the operation will be relevant for determining whether or not to select this operator. If the sea-state does not fall within an appropriate range (again, where the precise meaning of appropriate must be learned), the operation may fail.

The experiments to be generated should therefore vary the values of the wild cards (corresponding in this example to the *max-sea-state* value of *boom.1* and the *sea-state* in *sea-sector.1* at time *latest.1*), generate plans and plan evaluations to collect data about the success of the operation for different values of these wild cards, and construct hypotheses about what values are required for the operator to succeed.

To achieve this, each QC is matched against the initial world state. From the set of all possible matches (instantiations of the QC), a representative subset is selected by associating a value with each QC, consisting of a list of bindings for the wild cards in the QC. (Wild cards include the * argument and class names, but not variable names.) One instantiated QC is selected for each value that was seen.

For example, in *ml-get-boom1*, for the *sea-state* QC, the matches might be

```
(sea-state sf-bay 1 1)
(sea-state richards-bay 1 2)
(sea-state drakes-bay 1 1)
```

In this case, the values are 1, 2, and 1, respectively, and the first and second matches would be selected. For each of these matches, one or more constraints is identified that will result in this instantiation of the QC in a generated plan. For local variables (those whose value can be selected at the time the operator is applied), variable binding constraints are created for each constraint, For example, *boom.1* is not a local variable, so a variable binding is set up for each possible *max-sea-state* value associated with the available booms. For nonlocal variables, whose value is determined in a previously applied operator, the process is more complicated, and requires modifying the initial world state in such a way that the variable will be bound as desired. For example, *sea-sector.1* in *ml-get-boom1* is bound at a previous planning level to a location that is defined as a sensitive area in the planning scenario. We have not yet developed a general solution for computing the required modifications, so the user currently supplies the correct predicate(s) to establish for each QC. We plan to explore alternative methods for automating this process by analyzing the knowledge base and generated plans to trace the relevant dependencies to the critical decisions that cause the variables to be bound.

The full cross-product of experiments for each QC is formed, resulting in a list of combined experiments. (In principle, rather than computing the entire cross-product, individual combined experiments could be generated dynamically, by selecting and combining individual experiments for each QC.)

The Select Experiment subprocess then selects an experiment from the list. This can be done manually, or by selecting the first experiment on the list; an open problem is to explore methods for doing this automatically.

Data Generation

The Data Generation process uses the selected experiment to create the appropriate input data (training examples) for the Inductive Learning subprocess. This process incorporates three components: Generate Plan, Evaluate Plan, and Generate Training Example.

Generate Plan. A plan is generated interactively or automatically, using the constraints represented by the selected experiment. The constraints that can be specified in an experiment are

Variable Binding: bind a given variable to a given value in the specified operator.

Operator Selection: always select a specified operator when it is applicable.

Predicate Assertion: assert one or more predicates in the initial world state.

Object Creation: define a new object with specified properties before planning.

The implementation of these constraints uses SIPE-2's built-in hooks for variable binding of an operator selection during planning, and predicate/object definition, so it was straightforward to implement this module.

If other plans have already been generated using similar experiments, a limited capability exists to reuse those plans. We plan to expand this capability to use SIPE-2's replanning methods.

Evaluate Plan. The plan is sent to the evaluation model, which returns an evaluation indicating the success or failure of the overall plan and of individual actions or action sequences in the plan. The definition of success depends on the domain and on the purpose of the operator being learned (e.g., whether a unit arrived on time, or oil was successfully cleaned up).

One difficulty we had is that there is not a direct mapping in the oilspill domain between the purpose of a given operator and the quantities returned by the evaluation model. There are three primary reasons for this: first, multiple operators are often applied to achieve a single shared goal, so credit assignment is a problem. For example, in the *ml-get-boom1* operator, the purpose is to get a certain quantity of boom to the location of the operation; this can be done by combining several operators that each bring a smaller quantity of boom to the final destination. Therefore, the actions in a given operator may succeed while its overall purpose still fails.

Second, the purpose in the SIPE-2 sense may not match the conceptual purpose of the operator. For the *ml-get-boom1* operator, although the purpose slot lists the boom level as the operator's purpose, a planning expert would say that the real purpose of this operator is to contain a certain quantity of oil. Although the operator clearly would fail if the boom did not arrive in the given location by the required time, the operator would also fail if the boom did not work properly in the prevailing ocean conditions; this is not explicitly stated in the purpose slot.

Finally, even if "containing the oil in the sector" were explicitly stated in the operator's purpose, that goal is not explicitly represented in the evaluation model. Rather, the output of the evaluation model specifies the quantity of oil in each sector at each point in time, as well as the quantities that ended up on shore, evaporated, or were transported out of the sector (e.g., via skimmer). "Containing the oil" thus must be translated into a quantitative measure of those values, which raises many questions about how to do this: How much oil must be contained (and by what time) for the operator to be considered successful? When oil is removed by skimmer and barge, so that no oil is left in the sector nor will it have escaped, should the operator be considered to succeed? If the oil sinks before it is contained, so that it cannot be cleaned up but the ocean surface is clear, is that a success?

Solving this problem will require better ontologies and reasoning methods for mapping between plans (and planning knowledge) and evaluation models. These will allow planning systems and evaluation models to interact and share knowledge effectively.

Generate Training Example. The Operator Learner next uses the qualitative constraints to generate a training example for the operator being learned, using the new plan and its evaluation. The training example consists of the relevant world state (i.e., any instantiations that match the qualitative constraints) at the point in the plan where the operator was applied, and a positive or negative label, depending on whether the operator succeeded or failed. An operator that generates several actions succeeds if all of the actions succeed; therefore, the preconditions for the operator will be the union of the preconditions learned from the training examples associated with each action within the operator.

The world state at the plan node where the operator was applied is generated by executing the plan up to that node. (The effects of this execution are undone after the training example is generated.) The qualitative constraints in the operator being learned determine which predicates and properties to extract from the world state, using the same matching techniques that were used to match the QCs against the initial world state during experiment generation.

Related Work

Gil (1992) describes research on experiment generation for knowledge acquisition in planners. The general approach is to identify missing preconditions by observing when actions fail, and then to generate experiments to determine the correct precondition. This work assumes a direct measure of the success of an action, a much simpler (completely transparent) domain representation, and no uncertainty in the domain.

Davis (1993) describes the use of metalevel knowledge identify and repair bugs in the knowledge base of TEIRESIAS, an expert system for stock market investment advising. Eshelman et al. (1993) describe MOLE, a system for knowledge acquisition for heuristic problem solving. MOLE generates an initial knowledge base interactively, and then detects and corrects problems by identifying "differentiating knowledge" that distinguishes among alternative hypotheses. Ginsberg et al. (1993) have developed SEEK, which performs knowledge base refinement by using a case base to generate plausible suggestions for rule refinement. Gil and Swartout (1994) have developed EXPECT, a knowledge acquisition architecture that dynamically forms expectations about the knowledge that needs to be acquired by the system, and then uses these expectations to interactively guide the user through the knowledge acquisition process.

Learning apprentices are a recent development in knowledge acquisition tools. Mitchell et al. (1993) characterize a learning apprentice as an "interactive, knowledge-based consultant" that observes and analyzes the problem-solving behavior of users. Mitchell et al. developed the LEAP apprentice that uses explanation-based learning (EBL) techniques to explain and generalize cases (traces of the user's problem-solving behavior) in the domain of digital circuits. DISCIPLE (Kodratoff & Tecuci 1993) also uses EBL, as well as similarity-based learning, to acquire problem-solving knowledge in the domain of design

for the manufacturing of loudspeakers. Our focus on inductive learning techniques differentiates our work from these systems.

Wang and Carbonell (1994) have developed a system that inductively learns planning control knowledge. Their system makes some simplifying assumptions (e.g., that there is no randomness, and that the system has a complete domain representation) that limit the applicability of their approach to complex, real-world domains.

Future Work

The architecture we have developed provides a framework and testbed in which we can begin to explore new approaches to the knowledge acquisition problem. In this section, we outline some possibilities for research directions in this area.

Other automated tools that generalize and simplify the knowledge base automatically would be useful. For example, two or more similar operators could be generalized by parametrizing constants or by adding conditional branches. Operators could be simplified syntactically by the removal of unused or redundant arguments, preconditions, and constraints. In some cases, operators could be merged in order to eliminate superfluous planning levels. Redundant or unused operators could be deleted.

The representations used within the Operator Learner as well as in the planner could be extended in several ways. Permitting more qualitative constraint types would require generalizing both the representation and the implementation of QCs. An improved ontology for plan quality would help in developing effective interfaces between the Operator Learner and evaluation models. Adding soft constraints to SIPE-2 would allow probabilistic learned knowledge to be incorporated more directly into the operators, increasing the correctness of the planning process.

More sophisticated techniques for experiment generation and selection would allow the system to use SIPE-2's replanning mechanisms when running multiple experiments, automatically identify modifications to the initial world state that are required to generate the desired training examples, and select an optimal sequence of experiments when computational resources are limited.

Conclusions

Automated and semi-automated tools for knowledge acquisition will become increasingly essential as large-scale planning systems are developed and deployed. Our research and application experience in this area have led us to the development of two such tools: an interactive graphical editor for developing planning operators and an inductive learning system that uses feedback from an evaluation model and the user's planning choices to refine and verify partial operators. These tools are currently being extended, and are actively being used in ongoing knowledge engineering for SOCAP. They provide an important foundation upon which to build large-scale, realistic planning applications. However, many open issues remain in the development of these tools, and this paper points out a number of concrete research directions in this area.

References

Desimone, R. V., and Agosta, J. M. 1993. Oil spill response simulation: The application of artificial intelligence planning techniques. In *Proceedings of the Simulation MultiConference.*

desJardins, M. 1992. *PAGODA: A Model for Autonomous Learning in Probabilistic Domains.* Ph.D. Dissertation, UC Berkeley. (Available as UCB CS Dept. Technical Report 92/678).

Kodratoff, Y., and Tecuci, G. 1993. Techniques of design and DISCIPLE learning apprentice. In *Readings in Knowledge Acquisition and Learning.* Morgan Kaufmann. 655–668.

Quinlan, J. R. 1993. *C4.5: Programs for Machine Learning.* Morgan Kaufmann.