An Analysis of ABSTRIPS

Craig A. Knoblock
University of Southern California
Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90292
knoblock@isi.edu

Abstract

ABSTRIPS [Sacerdoti, 1974] was the first system to automate the construction of abstraction hierarchies for planning. Despite the seminal nature of this work, the method ABSTRIPS uses to construct abstraction hierarchies is only described in vague terms, and there is no analysis of how the method works or when it will be effective. This paper fills this gap and presents a reconstruction and analysis of the algorithm used in ABSTRIPS. The analysis shows that the method for constructing abstractions implicitly assumes that the preconditions that are determined to be details will be independent. In those cases where the independence assumption fails to hold, ABSTRIPS can degrade the performance of the planner. The paper also compares the ABSTRIPS approach to generating abstractions to the one used in ALPINE [Knoblock, 1990] and describes how ALPINE avoids the problem that arises in ABSTRIPS.

1 Introduction

One approach to reducing search in planning is to exploit abstractions of a problem space to plan hierarchically. A problem is first solved in an abstract space and then refined at successively more detailed levels. The ABSTRIPS system [Sacerdoti, 1974] provided some of the earliest work on abstraction in planning. ABSTRIPS was built on top of the STRIPS planning system [Fikes and Nilsson, 1971] and would first construct an abstraction hierarchy for a problem space and then use the abstractions for hierarchical planning. Hierarchical planning was pioneered in GPS [Newell and Simon, 1972], but ABSTRIPS was the first system that automated the construction of abstraction hierarchies.

In ABSTRIPS, the abstraction spaces are constructed by assigning *criticalities*, numbers indicating relative difficulty, to the preconditions of each operator. The system uses these criticalities to plan abstractly. First, an abstract plan is found that satisfies only the preconditions of the operators with the highest criticality values. The abstract plan is then refined by considering the preconditions at the next level of criticality and inserting steps into the plan to achieve these preconditions. The process is repeated until the plan is expanded to the lowest level of criticality.

The work on ABSTRIPS is widely viewed as seminal work on abstraction in planning [Korf, 1987, Tenenberg, 1988]. Yet, to date there has been no detailed description or analysis of the method used in ABSTRIPS for constructing abstraction hierarchies. In addition, recent results presented in [Knoblock, 1991] indicate that the abstraction hierarchies generated by ABSTRIPS actually degrade performance rather than improve it when used in the PRODIGY problem solver [Minton *et al.*, 1989, Carbonell *et al.*, 1991]. This paper presents a detailed analysis of how ABSTRIPS works. It also explains the apparent contradiction in that the abstractions generated by ABSTRIPS worked well in the STRIPS system, yet produced poor performance when used in PRODIGY.

The remainder of this paper is organized as follows. The next section presents a rational reconstruction of the algorithm for generating abstractions and an analysis of how and when this approach will work. The third section presents an example problem that illustrates a shortcoming of the ABSTRIPS approach. The fourth section compares the approaches used in ABSTRIPS and ALPINE [Knoblock, 1990, Knoblock, 1991] for generating abstractions and shows how ALPINE avoids the problem that arises in ABSTRIPS. The fifth section presents experimental results that compare ABSTRIPS' abstractions to ALPINE's in the PRODIGY problem solver. The sixth section compares the criteria used for generating abstractions in ABSTRIPS to the criteria used in other systems, and the last section summarizes the findings presented in this paper. In addition, the Lisp code for the ABSTRIPS algorithm is included in Appendix A, and the definition of the STRIPS robot planning domain is provided in Appendix B.

2 Algorithm and Analysis of ABSTRIPS

ABSTRIPS is given a definition of a problem space and a partial order of predicates, and it forms abstract problem spaces by assigning criticalities to the preconditions of the operators. According to [Sacerdoti, 1974], the criticalities are assigned as follows:

> A predetermined (partial) ordering of all the predicates used in describing the problem domain was used to specify an order for examining the literals of the precondition wffs of all the operators in the domain. First, all literals whose truth value could not be changed by an operator in the domain were assigned a maximum criticality value. Then, each remaining literal was examined in an order determined by the partial ordering. If a short plan could be found to achieve a literal from a state in which all previously processed literals were assumed to be true, then the literal in question was said to be a detail and was assigned a criticality equal to its rank in the partial ordering. If no such plan could be found, the literal was assigned a criticality greater than the highest rank in the partial order.

The crucial point left unspecified in the description of the algorithm is how the system determines whether a given precondition can be achieved by a short plan. To understand this point, I constructed a complete version of the algorithm that matches the description in the original paper and produces the same criticality assignments as those published on ABSTRIPS. The core of the algorithm is presented in Table 1 and the complete program is provided in Appendix A. The algorithm described below appears to be quite close to the original since it produces the same criticality assignments for the STRIPS domain, requires the same axioms that ABSTRIPS required to produce these criticalities, and is quite simple.

The algorithm presented in Table 1 assigns a criticality value to a precondition of an operator by attempting to prove that for any instantiation of the precondition, there exists a sequence of operators that will achieve the precondition. It is unclear what was meant by a short plan in the description above. The algorithm presented here considers a short plan to be any nonrecursive sequence of operators.

The function, `assign-precond-crit`, is given an operator and a precondition of that operator and assigns a criticality to that precondition. If the precondition cannot be changed by any operator it is *static* and is assigned a criticality of two plus the maximum value in the partial order. If it cannot show that a plan exists to achieve the precondition it is assigned a criticality

Table 1: Algorithm for Assigning Criticalities

```
function assign-precond-crit (precond,operator)
 begin
 if precond is static
 then return(2 + maximum value in partial order)
 else
   begin
   context ← context(precond,operator,[]);
   if not plan-exists(precond,context,operator)
   then return(1 + maximum value in partial order)
   else return(level of precond in partial order)
   end
 end

function context (goal,operator,prev-context)
 begin
 new-context ←  preconditions of operator higher
                than goal in the partial order;
 derived-context ←  conditions derived from axioms,
                    goal, and new-context;
 return(new-context ∪ derived-context
                    ∪ prev-context)
 end

function plan-exists (goal,context,stack)
 begin
 relevant-ops ← operators relevant to goal;
 if there-exists op ∈ relevant-ops such-that
     begin
     preconds ←  preconditions of op;
     and op ∉ stack;
         achievable(preconds,context,op,stack)
     end
 then return(true)
 else return(false)
 end

function achievable (preconds,context,op,stack)
 begin
 if forall pre ∈ preconds
   begin
   new-context ← context(pre,op,context);
   new-stack ← push(op,stack);
   or pre ∈ context;
      plan-exists(pre,new-context,new-stack)
   end
 then return(true)
 else return(false)
 end
```

of one plus the maximum value in the partial order. If a plan does exist, it is assigned a criticality equal to its level in the partial order.

Before determining whether a plan exists, the function `context` is called first to determine the context of the precondition. The context consists of the conditions that will hold at the time the precondition arises as a goal. These conditions consist of the other preconditions of the operator that are higher in the partial order than the precondition under consideration, any conditions that can be derived from these conditions using the axioms, and any previous context or conditions that held up to this point.

Given the context, the function `plan-exists` attempts to show that a plan will always exist for achieving the given precondition. It first determines the set of relevant operators for achieving the precondition and then tests each operator to see if its preconditions will be achievable in the given context. The algorithm avoids recursing infinitely by keeping a stack of the operators under consideration and failing whenever it encounters an operator that is already on the stack.

The function `achievable` determines whether a set of preconditions will be achievable in the given context. To do this, the function attempts to show that for each of the preconditions, the precondition either holds in the given context or a plan exists to achieve the precondition. In the later case, the function `plan-exists` is called recursively on the precondition, first updating the context and operator stack with the operator under consideration.

The essence of this approach is to determine whether a plan will always exist to achieve a given precondition. If such a plan exists, the precondition is considered a detail and can be separated from the more difficult aspects of the problem. Thus, the system *automatically* produces only a three-level abstraction hierarchy, with the static literals at the top of the hierarchy, the difficult to achieve literals next, and the details at the bottom. The user-defined partial order is then used to partition the details into separate levels.

For the problems tried in the ABSTRIPS paper this algorithm for generating abstractions appears to work very well. However, the algorithm assumes that the different preconditions or goal conditions will not interact with one another. This assumption arises because the algorithm only considers each of the preconditions independently. In practice, a plan for achieving a subgoal may interact with a plan for achieving another subgoal. Even in the STRIPS domain there are combinations of goal conditions where this type of interaction occurs. The effect of such an interaction is that a precondition that appeared to be achievable in the analysis, might not be achievable because the plan for achieving another goal could interact with it.

3 An Example Problem

Consider an example that illustrates the problem that can occur in ABSTRIPS when the independence assumption is violated. The problem is from the STRIPS robot planning domain [Fikes and Nilsson, 1971] and was selected from the randomly generated problems used in the experiments described later in this paper. This domain is defined in Appendix B.

The problem involves achieving the following conjunction of five goals:

```
(and (inroom a room1)
     (status door56 closed)
     (status door12 closed)
     (inroom robot room3)
     (inroom b room6)),
```

where the initial state for this problem is shown in Figure 1. The complete specification of this problem is included in Appendix D of [Knoblock, 1991].

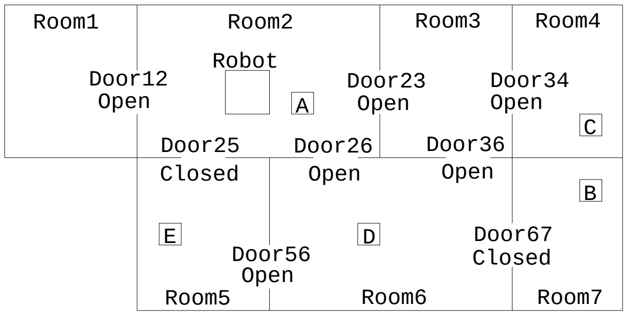

Figure 1: Initial State for the Example Problem

The criticality assignments generated by ABSTRIPS for this problem domain are specified in [Sacerdoti, 1974] and shown in Appendix B. ABSTRIPS assigns the static preconditions the highest criticality, 6. It assigns the preconditions that cannot be achieved by a short plan a criticality of 5. These preconditions include all of the `inroom` conditions and the `status` and `nextto` preconditions of the operators for opening and closing doors. The `status` and `nextto` preconditions of the remaining operators are shown to be details and are assigned criticalities that correspond to their values in the partial order given to ABSTRIPS, which is 2 for the `status` preconditions and 1 for the `nextto` preconditions.

Problem solving using this abstraction hierarchy proceeds as follows. Since ABSTRIPS only drops preconditions, all the top-level goals are considered in the abstract space. The system constructs an abstract plan to move box `a` into `room1`, closes the door to the room, and then moves the robot through the closed door in order to achieve the remaining goals. When the system is planning at this abstraction level it ignores all preconditions involving door status, so it does not notice that it will later have to open this door to make the plan work. When the plan is refined to the next level of detail, the steps are added to open the door before moving the robot through the door, deleting a condition that was achieved in the abstract space. At this point the problem solver would need to either backtrack or insert additional steps for closing the door again. The original ABSTRIPS system would not have even noticed that it had violated the precondition, and would simply produce an incorrect plan

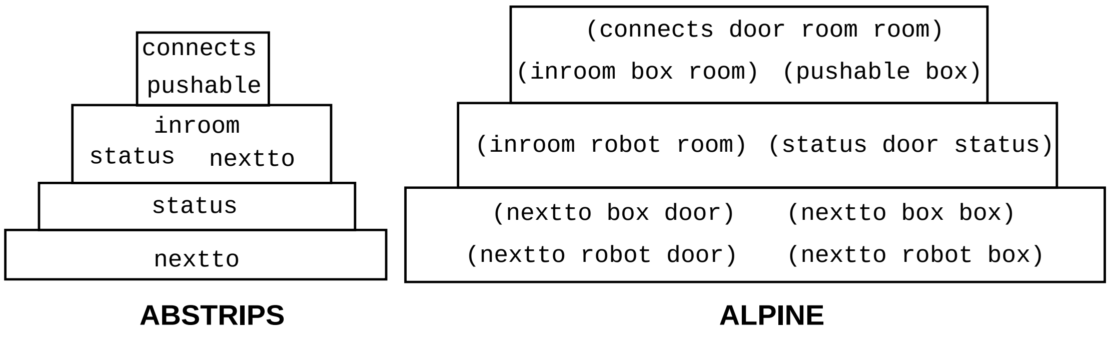
connects
pushable
inroom
status nextto
status
nextto
ABSTRIPS
(connects door room room)
(inroom box room) (pushable box)
(inroom robot room) (status door status)
(nextto box door) (nextto box box)
(nextto robot door) (nextto robot box)
ALPINE

Figure 2: Abstraction Hierarchies Generated by ABSTRIPS and ALPINE

[Joslin and Roach, 1989, pg.100]. In the analysis used by ABSTRIPS, the door status appeared to be a detail, but during planning considerable time can be wasted before it finally backtracks to the correct point in the search space and corrects the mistake.

4 Comparison with ALPINE

The ALPINE system [Knoblock, 1990, Knoblock, 1991] completely automates the generation of abstraction hierarchies from the definition of a problem space. Each abstraction space in a hierarchy is formed by dropping literals (atomic formulas) from the original problem space, thus it abstracts the preconditions and effects of operators as well as the states and goals of a problem space. ALPINE forms abstraction hierarchies based on the *ordered monotonicity* property [Knoblock *et al.*, 1991], which requires that the truth value of a literal introduced at one level is not changed at a lower level. This property guarantees that the preconditions that are achieved in an abstract plan will not be deleted (clobbered) while refining that plan. To construct abstraction hierarchies with this property, ALPINE analyzes the preconditions and effects of the operators to determine potential interactions. If a plan for achieving literal C_1 can change the truth value of literal C_2, then literal C_1 cannot be in a lower abstraction level than literal C_2. The detailed algorithm is described in [Knoblock, 1991].

ALPINE's approach to generating abstractions differs from ABSTRIPS in several important ways. First, ALPINE forms abstractions based on the ordered monotonicity property, while ABSTRIPS forms abstractions by considering whether each precondition of each operator can be achieved by a short plan. Second, ALPINE completely automates the construction of the abstraction hierarchies using the definition of the problem space, while ABSTRIPS requires an initial partial order on the predicates to form the abstraction hierarchy. Third, ALPINE forms abstractions that are tailored to each problem, while ABSTRIPS constructs a single abstraction hierarchy for the entire domain. Fourth, ALPINE forms *reduced models* where each level in the abstraction hierarchy is an abstraction of the original problem space, while ABSTRIPS forms *relaxed models* where only the preconditions are dropped.

The abstraction hierarchies generated by each system for the example problem are shown in Figure 2. As described earlier, ABSTRIPS uses the same four-level abstraction hierarchy for the entire problem domain. The most abstract space consists of all the static preconditions (the preconditions that cannot be changed), the second level consists of the preconditions that cannot be achieved by a short plan. This includes all of the **inroom** preconditions, and some of the **nextto** and **status** preconditions. The third level consists of the **status** preconditions that can be achieved by a short plan, and the fourth level contains the remaining **nextto** conditions that can also be achieved by a short plan.

The abstraction hierarchy built by ALPINE for this problem consists of the three-level abstraction hierarchy shown in Figure 2. (ALPINE builds finer-grained hierarchies by separating literals with the same predicate but different argument types.) The most abstract space consists of all the static literals and the **(inroom box room)** literals. The next level contains both the **(inroom robot room)** and the **(status door status)** literals. These two sets of literals are placed at the same level in order to satisfy the ordered monotonicity property since it may be necessary to get the robot into a particular room to open or close a door. Finally, the last level contains the **nextto** literals for both the robot and the boxes.

Using ALPINE's abstraction hierarchy to solve the example problem, a planner would first generate a plan for moving the boxes into the appropriate rooms. At the next level it would deal with the goals involving closing the doors and moving the robot. If it closed a door and then tried to move the robot through the door, it would immediately notice the interaction since these goals are considered at the same abstraction level. After producing a plan at the intermediate level it would refine this plan into the ground space by inserting the remaining details, which consists of the conditions involving `nextto`.

In constructing the abstraction hierarchy to solve this problem, ALPINE recognized the potential interaction between the operator for achieving goals involving door status and the operators for achieving the other top-level goals, and it created an abstraction hierarchy where door status is not considered a detail. Given a different problem, for example, one where the only goal is to get the robot into a particular room, ALPINE would recognize that in that case door status is a detail and place it at a lower abstraction level.

5 Experimental Results

To illustrate the difference between ALPINE's and ABSTRIPS's abstractions, the use of these abstractions are compared using the PRODIGY problem solver [Carbonell *et al.*, 1991] in the STRIPS domain. This is not a completely fair comparison because the abstraction hierarchies generated by ABSTRIPS were intended to be used by the STRIPS problem solver. STRIPS employed a best-first search instead of a depth-first search, so the problem of expanding an abstract plan that is then violated during the refinement of that plan would probably be less costly. Nevertheless, the comparison emphasizes the difference between the abstraction hierarchies generated by ALPINE and ABSTRIPS and demonstrates the problem with the approach used to generate abstractions in ABSTRIPS.

First consider the results on the example problem described earlier. Table 2 shows the CPU time (in seconds), nodes searched, and solution length for each configuration. ALPINE significantly reduces both the search and the solution length compared to PRODIGY. In contrast ABSTRIPS takes almost six times longer than PRODIGY, although it does produce a solution of the same length as ALPINE. Note that since ALPINE must construct an abstraction hierarchy for each problem, the CPU times reported for ALPINE throughout this paper include the time to construct an abstraction hierarchy.

Table 2: Performance on the Example Problem

System	Time	Nodes	Length
Prodigy	14.5	259	25
Prodigy + Alpine	10.2	114	19
Prodigy + Abstrips	83.0	1,631	19

The graphs in Figure 3 compare the solution times and solution lengths of PRODIGY without using abstraction, PRODIGY using the abstractions produced by ABSTRIPS, and PRODIGY using the abstractions produced by ALPINE. Each configuration was run on 200 randomly generated problems in the STRIPS robot planning domain. PRODIGY was run in each configuration and given 600 CPU seconds to solve each of the problems. Out of the 200 problems, 197 of the problems were solvable in principle. The solution time graph in Figure 3 shows the average solution times for the 197 solvable problems. The solution length graph shows the average solution lengths on the 153 problems that were solved by all three configurations. In both graphs the problems are ordered by the shortest solution found by any of the configurations.

The graphs show that the use of ABSTRIPS' abstractions significantly degrades performance, while ALPINE's abstractions improve performance over the basic PRODIGY system. PRODIGY performs quite well on these problems even without abstraction. This is because the problem solver only needs to search a small portion of the search space since most mistakes can be undone by adding additional steps. Thus, on problem-solving time PRODIGY performed quite well, but it achieved this performance by trading solution quality. On the hardest set of problems, PRODIGY produces solutions that were on average fifty percent longer than ALPINE. In contrast, the use of ABSTRIPS' abstractions significantly increased the problem solving time, although they did improve the quality of the solutions. The reason for the poor performance using ABSTRIPS' abstractions is that the interactions between different subgoals in a problem generated many redundant choice points, which resulted in wasted effort in backtracking. In addition, the problems used in these experiments were much larger than the ones used in the original experiments and as a result these problems were much more likely to have interacting subproblems.

6 Related Work

The purpose of abstraction in planning is to solve the more difficult aspects of a problem first and then fill in the details. The general problem that arises in finding useful abstractions is determining which parts of a problem are details. ABSTRIPS defined the details to be those preconditions that are always achievable independent of the other subgoals in a problem. Thus the abstractions generated by ABSTRIPS consider the difficulty in achieving individual conditions, but do not take into account the potential interactions among different subgoals. This section compares the criterion used by ABSTRIPS for constructing abstractions to other closely related work.

PABLO [Christensen, 1991] is another system that generates abstractions for hierarchical planning. It uses a technique called *predicate relaxation* to determine the number of steps needed to achieve each predicate by partially evaluating the operators. This information is then used to focus the problem solver on the part of the problem that requires the greatest number of

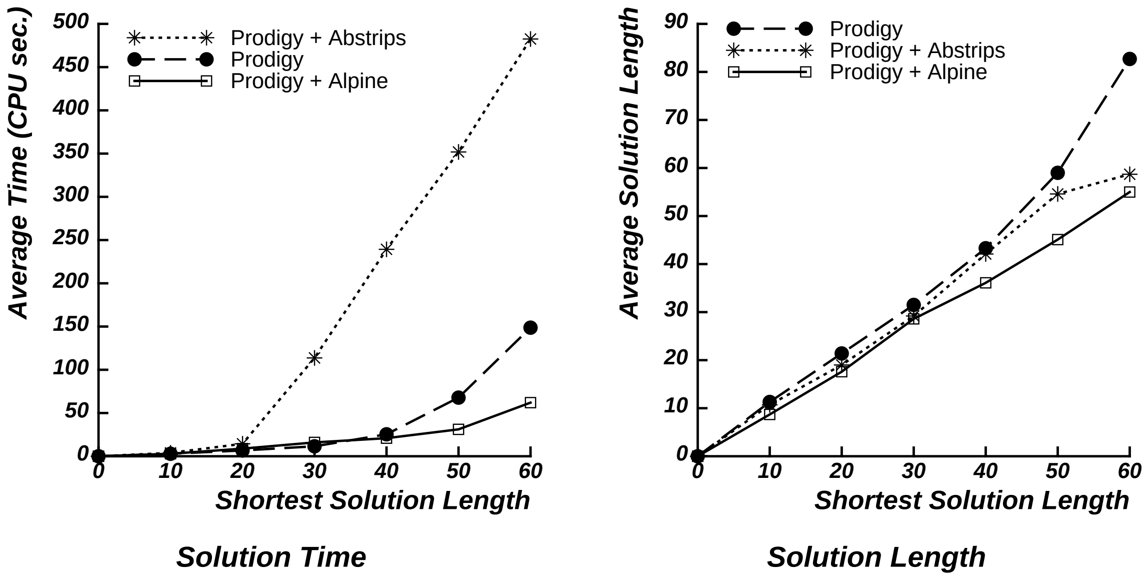

Figure 3: Comparison of the Average Solution Times and Average Solution Lengths

steps. The approach is quite similar to the one used by ABSTRIPS in that the abstractions are based on how many steps (in the worst case) it will take to achieve a given precondition instead of just whether or not a condition is always achievable. In is interesting to note that PABLO will suffer from exactly the same problem as ABSTRIPS because the predicate relaxation process does not consider interactions with plans for achieving other subgoals. Thus, due to interactions between subplans, it may believe that a subgoal is achievable when it is not, and it may underestimate the number of steps required to achieve a subgoal.

As described earlier in this paper, ALPINE forms abstractions based on the ordered monotonicity property. This property guarantees that any plan for achieving a condition ignored at an abstract level will not add or delete a literal in a more abstract space. In effect, the ordered monotonicity property partitions those conditions that interact with one another and orders the partitioned sets of conditions in a way that minimizes the interactions among them. While this property does consider the interactions between different goals, it does not guarantee that the conditions ignored at an abstract level will be achievable. However, neither ABSTRIPS or PABLO guarantee this property either since although they consider whether a individual precondition is achievable, they do not consider possible interactions that may preclude a solution.

Bacchus and Yang [1991] identified a stronger property called the *downward refinement property*, which states that if a problem is solvable then any abstract solution must have a refinement. Thus, if a solution to a problem exists, then the conditions ignored at the abstract level will be achievable. While clearly this is a desirable property, the problem is that this property can only be guaranteed in very restricted cases.

7 Conclusion

This paper presented a rational reconstruction of the algorithm used in ABSTRIPS for generating abstraction hierarchies. ABSTRIPS assigns criticalities to preconditions of operators by separating out those preconditions that can be shown to be always achievable. These preconditions are the details and achievement of them is delayed until the more difficult parts of the problem are solved. The flaw in this approach is that evaluating whether a precondition can be achieved is done in isolation from the rest of a problem. As shown in this paper, this can lead to performance that is worse than no abstraction at all when there are interactions between plans for achieving different subgoals. From this we can conclude that the problem spaces in which ABSTRIPS will be effective are those in which there are preconditions that can be shown to be achievable and those preconditions can be achieved independently.

The approach to generating abstractions in ABSTRIPS was compared to the approach in ALPINE. ALPINE produces abstraction hierarchies that have the ordered monotonicity property, which guarantees that the goals and subgoals that arise in the process of refining an abstract plan will not interact with the conditions already achieved in a more abstract level. Unlike ABSTRIPS, ALPINE considers all of the potential interactions in a problem space and avoids precisely the type of interactions that lead to poor performance in ABSTRIPS. As a result, ALPINE was able to improve the performance of PRODIGY, while ABSTRIPS degraded it.

A Program for Assigning Criticalities

```
; Top-level function for assigning criticalities.
(defun abstrips (&optional (ops *OPERATORS*))
  (load-operators ops)
  (calc-max-value-in-po)
  (assign-criticalities ops))

; Process each operator in the problem space.
(defun load-operators (ops)
  (cond ((null ops))
        (t (load-operator (caar ops)(cdar ops))
           (load-operators (cdr ops)))))

; Store preconds and effects on property list.
(defun load-operator (name body)
  (cond ((null body))
        (t (setf (get name (caar body))
                 (cadar body))
           (load-operator name (cdr body)))))

; Maximum value in the partial order.
(defun calc-max-value-in-po ()
  (setq *MAX-VALUE-IN-PO*
   (apply #'max (mapcar #'cdr *PARTIAL-ORDER*))))

; Assign criticalities to each operator.
(defun assign-criticalities (&optional
                             (ops *OPERATORS*))
  (cond ((null ops))
        (t (format t "~%~%~a" (caar ops))
           (assign-op-crits
            (caar ops)(preconds (caar ops)))
           (assign-criticalities (cdr ops)))))

; Assign criticalities to each precondition.
(defun assign-op-crits (op preconds)
  (cond ((null preconds))
        (t (format t "~%   ~a ~a"
                   (assign-precond-crit
                    (car preconds) op)
                   (car preconds))
           (assign-op-crits op (cdr preconds)))))

; Assigns criticality based on whether the
; precondition is static and whether or not a
; plan always exists to achieve the precondition.
(defun assign-precond-crit (precond operator)
   (cond ((static precond)
          (+ 2 *MAX-VALUE-IN-PO*))
         ((not (plan-exists
                precond
                (context precond
                         (preconds operator)
                         nil)
                (relevant-ops precond)
                (list operator)))
          (+ 1 *MAX-VALUE-IN-PO*))
         (t (level-in-po precond))))

; A plan exists to achieve a goal if the
; preconditions of any of the relevant operators
; are achievable. Maintains a stack of operators
; under consideration to avoid looping.
(defun plan-exists (goal context relevant-ops
                         op-stack)
  (cond ((null relevant-ops) nil)
        ((member (car relevant-ops) op-stack)
         (plan-exists goal context
                      (cdr relevant-ops)
                      op-stack))
        ((achievable (preconds
                      (car relevant-ops))
                     context
                     (car relevant-ops)
                     op-stack))
        (t (plan-exists goal context
                        (cdr relevant-ops)
                        op-stack))))

; A set of preconditions is achievable if each
; precondition is either true in the given context or
; it can be shown that a plan exists to achieve it.
(defun achievable (preconds context relevant-op
                            op-stack)
  (cond
    ((null preconds))
    ((or (member (car preconds) context
                 :test #'matches)
         (plan-exists
          (car preconds)
          (context (car preconds)
                   (preconds relevant-op)
                   context)
          (relevant-ops (car preconds))
          (cons relevant-op op-stack)))
     (achievable (cdr preconds) context
                 relevant-op op-stack))))

; Given both a goal and the preconditions of
; an operator that are relevant to achieving that
; goal, the context consists of all those preconditions
; that are higher in the partial order than the goal,
; the conditions that in turn can be derived from those
; preconditions, and the conditions in the previous
; context in which the goal arose.
(defun context (goal preconds prev-context)
  (let ((new-context (higher-in-po
                      (level-in-po goal)
                      preconds)))
    (append (derive-context goal new-context)
            new-context
            prev-context)))

; Level in the partial order of a given goal.
(defun level-in-po (cond)
  (cdr (assoc (car cond) *partial-order*)))

; Finds conditions higher in the partial order
; than the given criticality.
(defun higher-in-po (crit list)
  (cond ((null list) nil)
        ((<= (level-in-po (car list)) crit)
         (higher-in-po crit (cdr list)))
        (t (cons (car list)
                 (higher-in-po crit
                               (cdr list))))))
```

```
; The derived context consists of those conditions
; that can be derived from the negation of the
; goal (the negation of the goal must hold or it
; wouldn't be a goal) and from the given context.
(defun derive-context (goal context)
  (append (derive (list `not goal))
          (derived-context context)))

; A condition can be derived from the context if
; it can be derived from an atomic formula in the
; context.
(defun derived-context (context)
  (cond ((null context) nil)
        (t (append (derive (car context))
                   (derived-context
                    (cdr context))))))

; Attempts to match a condition against the
; left-hand side of the axioms.  If it matches,
; the right-hand side of the axiom holds.
(defun derive (cond)
  (cdr (assoc cond *AXIOMS* :test #'matches)))

; A precondition is static if there are no
; operators relevant to achieving it.
(defun static (condition)
  (cond ((null (relevant-ops condition)))))

; An operator is relevant to achieving a goal if
; it is in the primary effects of the operator.
(defun relevant-ops (condition)
  (cdr (assoc condition *PRIMARY*
              :test #'matches)))

; Two literals match if the constants match
; and the variables can be bound appropriately.
; Note that no substitution list is maintained, so
; this function could incorrectly determine that
; two literal match.
(defun matches (cond1 cond2)
  (cond ((and (null cond1)(null cond2)))
        ((or (null cond1)(null cond2)) nil)
        ((listp (car cond1))
         (if (matches (car cond1)(car cond2))
             (matches (cdr cond1)(cdr cond2))))
        ((and (not (is-variable (car cond1)))
              (not (is-variable (car cond2)))
              (not (eq (car cond1)(car cond2))))
         nil)
        (t (matches (cdr cond1)(cdr cond2)))))

; Returns a list of preconditions of an operator.
(defun preconds (operator)
  (let ((preconds (get operator 'preconds)))
    (if (eq 'and (car preconds))
        (cdr preconds)
        preconds)))

; A variable begins with a '<'.
(defun is-variable (atm)
   (and (symbolp atm)
        (eql '#\< (char (symbol-name atm) 0))))
```

B Strips Robot Planning Domain

```
; An equivalent version of this domain first
; appeared in [Sacerdoti 1974].

; Initial partial order of the predicates.
(defparameter *PARTIAL-ORDER*
  '((type . 4)(connects . 4)
    (locinroom . 4)(pushable . 4)
    (inroom . 3)
    (status . 2)
    (nextto . 1)))

; Axioms that state invariant properties of the
; domain.  The first axiom states that if something
; is pushable, then it is of type object.  The
; second axiom states that if a door is not open,
; then it is closed.
(defparameter *AXIOMS*
 '(((pushable <x>)(type <x> object))
   ((not (status <x> open))(status <x> closed))))

; Specifies which operators have a literal as
; a primary add.
(defparameter *PRIMARY*
 '(((nextto robot <object>) . (GOTO-BOX
                               GOTO-DOOR))
   ((at robot <loc> <loc>) . (GOTO-LOC))
   ((nextto <object> <object>) . (PUSH-BOX
                                  PUSH-TO-DOOR))
   ((at <object> <loc> <loc>) . (PUSH-TO-LOC))
   ((inroom robot <room>) . (GO-THRU-DOOR))
   ((inroom <object> <room>) . (PUSH-THRU-DOOR))
   ((status <door> open) . (OPEN-DOOR))
   ((status <door> closed) . (CLOSE-DOOR))))

; The operators are defined in the PRODIGY
; language [Minton et al, 1989], so the syntax
; differs slightly from the original.  The
; criticalities assigned by ABSTRIPS are
; shown in curly braces.
(defparameter *OPERATORS* '(

(GOTO-BOX
 (params (<bx>))
 (preconds
  (and (type <bx> object) ;{6}
       (inroom <bx> <rx>) ;{5}
       (inroom robot <rx>))) ;{5}
 (effects ((del (at robot <loc1> <loc2>))
           (del (nextto robot <obj1>))
           (add (nextto robot <bx>)))))

(GOTO-DOOR
 (params (<dx>))
 (preconds
  (and (type <dx> door) ;{6}
       (inroom robot <rx>) ;{5}
       (connects <dx> <rx> <ry>))) ;{6}
 (effects
  ((del (at robot <loc1> <loc2>))
   (del (nextto robot <obj1>))
   (add (nextto robot <dx>)))))
(GOTO-LOC
 (params (<x> <y>))
 (preconds
  (and (inroom robot <rx>) ;{5}
       (locinroom <x> <y> <rx>))) ;{6}
 (effects
  ((del (at robot <loc1> <loc2>))
   (del (nextto robot <obj1>))
   (add (at robot <x> <y>)))))

(PUSH-BOX
 (params (<bx> <by>))
 (preconds
  (and (type <by> object) ;{6}
       (pushable <bx>) ;{6}
       (nextto robot <bx>) ;{1}
       (inroom <bx> <rx>) ;{5}
       (inroom <by> <rx>) ;{5}
       (inroom robot <rx>))) ;{5}
 (effects
    ((del (at robot <loc1> <loc2>))
     (del (nextto robot <obj1>))
     (del (at <bx> <loc3> <loc4>))
     (del (nextto <bx> <obj2>))
     (del (nextto <obj3> <bx>))
     (add (nextto <by> <bx>))
     (add (nextto <bx> <by>))
     (add (nextto robot <bx>)))))

(PUSH-TO-DOOR
 (params (<bx> <dx>))
 (preconds
  (and (pushable <bx>) ;{6}
       (type <dx> door) ;{6}
       (nextto robot <bx>) ;{1}
       (inroom robot <rx>) ;{5}
       (inroom <bx> <rx>) ;{5}
       (connects <dx> <rx> <ry>))) ;{6}
 (effects
    ((del (at robot <loc1> <loc2>))
     (del (nextto robot <obj1>))
     (del (at <bx> <loc3> <loc4>))
     (del (nextto <bx> <obj2>))
     (del (nextto <obj3> <bx>))
     (add (nextto <bx> <dx>))
     (add (nextto robot <bx>)))))

(PUSH-TO-LOC
 (params (<bx> <x> <y>))
 (preconds
   (and (pushable <bx>) ;{6}
        (nextto robot <bx>) ;{1}
        (inroom robot <rx>) ;{5}
        (inroom <bx> <rx>) ;{5}
        (locinroom <x> <y> <rx>))) ;{6}
 (effects
    ((del (at robot <loc1> <loc2>))
     (del (nextto robot <obj1>))
     (del (at <bx> <loc3> <loc4>))
     (del (nextto <bx> <obj2>))
     (del (nextto <obj3> <bx>))
     (add (at <bx> <x> <y>))
     (add (nextto robot <bx>)))))

(GO-THRU-DOOR
 (params (<dx> <rx>))
 (preconds
  (and (type <dx> door) ;{6}
       (type <rx> room) ;{6}
       (status <dx> open) ;{2}
       (inroom robot <ry>) ;{5}
       (connects <dx> <ry> <rx>))) ;{6}
 (effects
    ((del (at robot <loc1> <loc2>))
     (del (nextto robot <obj1>))
     (del (inroom robot <ry>))
     (add (inroom robot <rx>)))))

(PUSH-THRU-DOOR
 (params (<bx> <dx> <rx>))
 (preconds
  (and (pushable <bx>) ;{6}
       (type <dx> door) ;{6}
       (type <rx> room) ;{6}
       (status <dx> open) ;{2}
       (nextto <bx> <dx>) ;{1}
       (nextto robot <bx>) ;{1}
       (inroom <bx> <ry>) ;{5}
       (inroom robot <ry>) ;{5}
       (connects <dx> <ry> <rx>))) ;{6}
 (effects
    ((del (at robot <loc1> <loc2>))
     (del (nextto robot <obj1>))
     (del (at <bx> <loc3> <loc4>))
     (del (nextto <bx> <obj2>))
     (del (nextto <obj3> <bx>))
     (del (inroom robot <ry>))
     (del (inroom <bx> <ry>))
     (add (inroom <bx> <rx>))
     (add (inroom robot <rx>))
     (add (nextto robot <bx>)))))

(OPEN-DOOR
 (params (<dx>))
 (preconds
  (and (type <dx> door) ;{6}
       (status <dx> closed) ;{5}
       (nextto robot <dx>))) ;{5}
 (effects
    ((del (status <dx> closed))
     (add (status <dx> open)))))

(CLOSE-DOOR
 (params (<dx>))
 (preconds
  (and (type <dx> door) ;{6}
       (status <dx> open) ;{5}
       (nextto robot <dx>))) ;{5}
 (effects
    ((del (status <dx> open))
     (add (status <dx> closed)))))
))
```

Acknowledgments

I would like to thank Yigal Arens, Oren Etzioni, Soowon Lee, Ramesh Patil, Paul Rosenbloom, and Tom Russ for their comments and suggestions on various drafts of this paper.

The research reported here was supported in part by an Air Force Laboratory Graduate Fellowship through the Human Resources Laboratory at Brooks AFB, in part by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U.S. Air Force, Wright-Patterson AFB, OH 45433-6543 under Contract F33615-90-C-1465, Arpa Order No. 7597, and in part by Rome Laboratory of the Air Force Systems Command and the Defense Advanced Research Projects Agency under contract no. F30602-91-C-0081. Views and conclusions contained in this report are the author's and should not be interpreted as representing the official opinion or policy of DARPA, HRL, AL, RL, the U.S. Government, or any person or agency connected with them.

References

[Bacchus and Yang, 1991] Fahiem Bacchus and Qiang Yang. The downward refinement property. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, pages 286–292, Sydney, Australia, 1991.

[Carbonell *et al.*, 1991] Jaime G. Carbonell, Craig A. Knoblock, and Steven Minton. PRODIGY: An integrated architecture for planning and learning. In Kurt VanLehn, editor, *Architectures for Intelligence*, pages 241–278. Lawrence Erlbaum, Hillsdale, NJ, 1991. Available as Technical Report CMU-CS-89-189.

[Christensen, 1991] Jens Christensen. *Automatic Abstraction in Planning*. Ph.D. Thesis, Department of Computer Science, Stanford University, 1991.

[Fikes and Nilsson, 1971] Richard E. Fikes and Nils J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[Joslin and Roach, 1989]
David Joslin and John Roach. A theoretical analysis of conjunctive-goal problems. *Artificial Intelligence*, 41(1):97–106, 1989.

[Knoblock *et al.*, 1991] Craig A. Knoblock, Josh D. Tenenberg, and Qiang Yang. Characterizing abstraction hierarchies for planning. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, Anaheim, CA, 1991.

[Knoblock, 1990] Craig A. Knoblock. Learning abstraction hierarchies for problem solving. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 923–928, Boston, MA, 1990.

[Knoblock, 1991] Craig A. Knoblock. *Automatically Generating Abstractions for Problem Solving*. Ph.D. Thesis, School of Computer Science, Carnegie Mellon University, 1991. Available as Technical Report CMU-CS-91-120.

[Korf, 1987] Richard E. Korf. Planning as search: A quantitative approach. *Artificial Intelligence*, 33(1):65–88, 1987.

[Minton *et al.*, 1989] Steven Minton, Jaime G. Carbonell, Craig A. Knoblock, Daniel R. Kuokka, Oren Etzioni, and Yolanda Gil. Explanation-based learning: A problem solving perspective. *Artificial Intelligence*, 40(1-3):63–118, 1989.

[Newell and Simon, 1972] Allen Newell and Herbert A. Simon. *Human Problem Solving*. Prentice-Hall, Englewood Cliffs, NJ, 1972.

[Sacerdoti, 1974] Earl D. Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5(2):115–135, 1974.

[Tenenberg, 1988] Josh D. Tenenberg. *Abstraction in Planning*. Ph.D. Thesis, Computer Science Department, University of Rochester, 1988.