# A Problem-Solver for Making Advice Operational

Jack Mostow
USC Information Sciences Institute
4676 Admiralty Way.
Marina del Rey, CA. 90291[1]

## Abstract

One problem with taking advice arises when the advice is expressed in terms of data or actions unavailable to the advice-taker. For example, in the card game Hearts, the advice "don't lead a suit in which some opponent has no cards left" is non-operational because players cannot see their opponents' cards. *Operationalization* is the process of converting such advice into an executable (perhaps heuristic) procedure. This paper describes an interactive system, called BAR, that operationalizes advice by applying a series of program transformations. By applying different transformation sequences, BAR can operationalize the same advice in very different ways.

BAR uses means-ends analysis and planning in an abstraction space. Rather than using a hand-coded difference table, BAR analyzes the transformations to identify transformation sequences that might help solve a given problem. Thus new transformations can be added without modifying the problem-solver itself. Although user intervention is required to select among alternative plans, BAR reduces the number of alternatives by $10^3$ compared to an earlier operationalizer.

## 1. Introduction

Many tasks that are onerous to program seem much easier to specify in terms of a body of advice. Examples include air traffic control ("keep planes three miles apart"), factory scheduling ("minimize re-tooling"), document preparation ("place a figure on the page that mentions it"), and computer dating ("no incestuous matches"). The idea of the advice-taking machine has been around for some time [McCarthy 68]: in this paradigm, the machine accepts advice for how to perform a task and converts it into an effective procedure. [Hayes-Roth 81] explores this paradigm in some depth, showing how advice provided by an expert tutor could be refined by experience.

Several hard problems must be solved to achieve the ambitious long-term goal of a general advice-taker. One problem is to translate advice expressed imprecisely in natural language into a precise machine representation of its meaning. Another problem is to combine different, possibly conflicting pieces of advice. This paper focusses on a third problem: taking advice that is *non-operational*, i.e., expressed in terms of data or actions unavailable to the machine, and transforming it into a procedure executable using only the available operations. This process is called *operationalization* [Mostow 81] [Dietterich et al 82].

Operationalization cannot be studied in a vacuum: one must look at how advice is operationalized in the context of a particular task, ideally one that is easy to model but retains the essential properties of advice-based tasks: i.e., that no simple, economical algorithm is known (this rules out tasks like sorting), but good performance can be attained by following known advice (this rules out tasks like earthquake prediction). A task like air traffic control has both properties but is inconvenient to model. This paper uses the card game Hearts as its example domain, and is based on two programs, called FOO and BAR, that accept Hearts advice, encoded in a suitable internal representation, and operationalize it by applying a series of program transformations.

FOO [Mostow 81] was used to operationalize 13 pieces of advice for Hearts and a music composition task, including[2]:

- "Avoid taking a trick with points" — non-operational because a Hearts player cannot simply refuse to take points
- "Flush out the Queen of spades" — one player cannot choose the card played by another
- "Don't lead a high card in a suit where some opponent is void [has no cards left]" — a player may not peek at opponents' cards

By applying different sequences of transformations, the same piece of advice was operationalized in different ways. Thus "avoid taking points" was operationalized both as "play a low card" and as a heuristic search procedure that enumerates possible sequences of play for a trick to determine whether playing a given card might lead to taking points [Mostow 82]. "Flush the Queen" was operationalized as a plan to keep leading spades until whoever has the Queen is forced to play it. The problem of deciding whether some opponent is void in a given suit was operationalized in two ways. One way is to check if someone failed to follow suit earlier when that suit was led. Another is to

[1]This research was supported by DARPA Contract MDA-903-81-C-0335. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government. I am grateful to my dissertation committee (Rick Hayes-Roth, Allen Newell, Jaime Carbonell, and Bob Balzer) for manifold contributions to this research, to my ISI colleagues for their intellectual influence, and to Don Cohen and Bill Swartout for improving this paper.

[2]These pieces of advice were operationalized independently of each other, but in fact they interact. For example, the reason for flushing out the Queen of spades is to make someone else take it. Leading the King or Ace of spades might help flush out the Queen at the cost of taking it, which would violate the advice to avoid taking points.

estimate the probability that someone is void based on the number of cards played in the suit.

The examples handled by FOO averaged 60 steps in length. At each step, the decision of which transformation rule to apply and which part of the advice to apply it to were made by hand. Since FOO had 230 general transformation rules, the branching factor was on the order of $10^3$ to $10^4$. FOO's successor, BAR, helps automate the selection of rule sequences, reducing the branching factor to between 1 and 10 at each step. This $10^3$ improvement is achieved by a combination of means-ends analysis and abstract planning. BAR currently has 60 rules and handles about half of the 13 examples done using FOO.

## 2. A means-ends analysis approach

Both FOO and BAR incorporate a *problem space* [Newell 79] model of operationalization, whose states are expressions in a LISP-like language. For example, the expression (Void $p_0$ $s_0$) represents the condition "player $p_0$ is void in suit $s_0$." The operators in this space are general transformation rules.

BAR formalizes the goal structure left implicit in the transformation sequences generated using FOO. Goals in the operationalization space (as opposed to goals in the task domain itself) are expressed in a pattern language containing pattern variables, embedded tests, segments, Kleene star, and some other constructs.

A problem in this space consists of finding a sequence of transformation rules that rewrites a given expression to match a given pattern. This is accomplished by *means-ends analysis*, succinctly described in [Kant & Newell 82] as

> the continual comparison of the current state with the desired state (or its description); the result of the comparison (a difference or an opportunity) is used to select the next operator (to reduce the difference or exploit the opportunity).

Differences between an expression and a pattern include mismatches between corresponding symbols, argument transposition, and so forth; for a detailed description of BAR's pattern language and the kinds of differences, see [Mostow 83].

Two kinds of top-level operationalization problems are represented in the pattern language by the tests @IsEvaluable and @IsAchievable. The first represents the goal of figuring out how to evaluate a given expression in terms of observable data. The second represents the goal of finding executable actions to achieve a given condition. Subgoals arise when a selected transformation rule does not apply to the current expression; such a subgoal is represented by a pattern language description of the rule's left-hand side.[3]

Many means-ends problem-solvers have been built since GPS [Newell 60]; the next two sections describe the novel aspects of BAR.

[3] BAR's rules are encoded as procedures, but BAR reasons about them using a more convenient pattern language representation. The pattern language is powerful enough to capture most if not all of each rule.

## 3. Separation of knowledge

In BAR, different kinds of knowledge are factored apart and represented separately.

Task domain knowledge is encoded as concept definitions and features. For example, the predicate Void(player,suit) is defined as (Not (Exists card (Cards-in-hand player) (In-suit card suit))), and In-suit is marked IsPredicate and IsComputable to represent the fact that a Hearts player can test a given card to tell if it belongs to a given suit. BAR currently has 38 of FOO's 112 concept definitions.

Program transformations are encoded as rules that do not mention the task domain and may in fact be useful in more than one domain. BAR's 60 rules include unfolding and folding definitions [Darlington 76], approximating a predicate P probabilistically as (High (Pr P)) ("P is likely"), and using a combinatorial formula to compute the probability that two subsets randomly chosen from a given universe will be disjoint.

Knowledge about what is operational is encoded as general facts, such as "A computable function of evaluable arguments is evaluable," represented in BAR as (@IsComputable @IsEvaluable*) —matches→ @IsEvaluable.

Finally, knowledge about means-ends problem-solving is encoded procedurally. Thanks to this factoring of knowledge, domain knowledge and program transformations can be added to BAR without modifying the problem-solving procedure itself.

Unlike GPS, BAR uses no built-in difference table that indexes directly from differences to operators. Instead, BAR analyzes its transformations to determine what kinds of differences they might help reduce. Some of BAR's knowledge about analyzing transformations is illustrated below.

**"If the left side of a transformation rule contains an argument absent from the right side, the rule deletes the sub-expression corresponding to that argument."** For example, one such rule approximates a binary ordering as the corresponding unary predicate; this rule is described in the pattern language as (@IsOrdering ?x ?y) → (@IsPredicate ?x). BAR deduces that applying this rule to an expression has the effect of deleting the sub-expression bound to the pattern variable ?y. This knowledge is used in operationalizing the advice "avoid taking points" as "play a low card": to evaluate the expression (Lower (My-card) (Card-played-by ?q)) ("my card is lower than the card to be played by player ?q"), BAR decides to eliminate the unevaluable second argument by approximating the expression as (Low (My-card)).

**"If the right side of a rule occurs as a sub-pattern of the left side, the rule extracts the sub-expression corresponding to the sub-pattern."** Such a rule is useful when the current expression does not match the current goal but one of its sub-expressions does. One such rule is described in the pattern language as (@IsQuantifier ?x ?S [?P FreeOf ?x]) → ?P. This rule eliminates a quantifier when the quantified predicate is independent of (FreeOf) the quantified variable.

**"If both sides of a rule have the same top-most symbol, the rule restructures the expression to which it's applied."**

Such rules are useful when the arguments of the current expression cannot be rewritten to match the corresponding arguments of the current goal. One such rule transposes the arguments of a symmetric relation: (@IsSymmetric ?x ?y) → (@IsSymmetric ?y ?x).

**"If the right side of a rule is a constant, the rule evaluates the expression to which it's applied."** One such rule recognizes that a conjunction containing a false term is false: (And -- False --) → False. BAR knows that evaluation is one kind of simplification, and applies simplification rules whenever it can.

## 4. Abstract plans

BAR faces a combinatorially explosive problem space, involving transformation sequences dozens of steps long, with several rules applicable at each step. To cut down the combinatorics, BAR searches in a simpler abstracted space for plans to reduce differences. This restricts the set of transformations considered to those that lead to a solution in the abstracted version of the problem space. A transformation rule (L ...) → (R ...) or a fact (L ...) —matches→ (R ...) in the original space is abstracted as L → R in the simpler space. To rewrite an expression (f ...) to match a pattern (g ...), BAR searches in the abstracted space for paths of the form f —rule→ ... —rule→ g. Such paths are only a few steps long, and are found by depth-first search. Each such path constitutes an abstract plan for rewriting (f ...) to match (g ...).

In the course of applying such a plan, BAR must solve subproblems that arise when a proposed rule does not apply to the current expression. It does so by recursively invoking itself with the left side of the rule as the pattern to match. If the CAR (top-level function) of the expression matches the CAR of the pattern, BAR recursively transforms the arguments of the expression to match the arguments of the pattern; if this fails, it tries to restructure the expression, e.g., by transposing arguments, or looks for alternative plans to get to the goal. BAR's path-finder avoids returning more than one plan with a given first step, so a successful route to the goal may follow a plan part-way and then switch to a new plan in mid-stream.

When BAR finds only one plan, it applies it automatically, but when it finds more than one, it scores them based on such factors as the number of steps in the plan and how many of them require subgoaling. It then asks the user to select among them. Alternatively BAR can try them in order of decreasing score, but at present this tends to cause a runaway search.

Of course, an abstract plan may fail; this occurs when the current expression cannot be transformed to match the next step in the plan. To reduce the incidence of plan failure, BAR uses a simple form of learning by experience: it records how often it tries to match the left side of each rule, and how often it succeeds. One result discovered by this simple tuning mechanism is that it is easier to make an expression runtime-evaluable, i.e., to match the pattern @IsEvaluable, than to actually evaluate it, i.e., to match @IsConstant. This knowledge influences the order in which the abstracted space is searched, in the hope that of all the plans that start the same way, BAR will find the one most likely to succeed.

The combination of abstract planning and recursive descent means that BAR plans in a hierarchy of abstraction spaces [Sacerdoti 74] corresponding to successive nesting depths of sub-expressions.

## 5. A short example

This section shows how BAR operationalizes the condition "player $p_0$ is void in suit $s_0$." [Mostow 83] presents a longer example exhibiting complexities omitted here.

The initial problem is

**> Goal:**
**Rewrite (Void $p_0$ $s_0$) to match @IsEvaluable**

BAR finds 6 abstract plans to get from Void to @IsEvaluable, and rates them:

**Plan P1 (rated 226): Void -unfold→ Not -fold→ Disjoint —matches→ @IsEvaluable**

**Plan P2 (rated -91): Void —Rule227→ Implied-by —Fact4→ @IsEvaluable**

**Plan P3 (rated -167): Void —Rule234→ WasDuring —Rule227→ Implied-by —Fact4→ @IsEvaluable**

**Plan P4 (rated -108): Void —Rule193→ In —Rule173→ True —matches→ @IsEvaluable**

**Plan P5 (rated -127): Void —Rule405→ High —Fact4→ @IsEvaluable**

**Plan P6 (rated -240): Void —Rule329→ @IsQuantifier —Rule155→ @IsPredicate —Rule227→ Implied-by —Fact4→ @IsEvaluable**

In this example, the user selects plan P5. In more detail, plan P5 is

**1! Approximate predicate probabilistically using Rule405: ?P → (High (Pr ?P)).**

**2? Make the result evaluable.**

Plan P5 is rated low because Rule405 is marked as an approximating method. (Plan P1 leads to the same result by a different route; plans P2 and P3 lead to alternative solutions described later; P4 and P6 are dead ends.)

The "!" denotes the fact that step 1 can be applied immediately, producing the transformation

(Void $p_0$ $s_0$) → (High (Pr (Void $p_0$ $s_0$)))

The "?" denotes the fact that step 2 requires some subgoaling. The problem is that although the predicate High is marked IsComputable, there is no general definition for Pr. This problem leads to the subgoal

**> Goal:**
**Rewrite (Pr (Void $p_0$ $s_0$)) to match @IsEvaluable**

BAR finds one plan to get from Pr to @IsEvaluable and tries it without asking:

**2.1? Use formula for probability that two randomly chosen subsets of ?U will be disjoint:**
**(Pr (Disjoint (SetOf x ?U ?P) (SetOf y ?U ?Q)))**
**→ (Pr-disjoint-formula**
**(# (SetOf x ?U ?P))**
**(# (SetOf y ?U ?Q))**
**(# ?U))**

**2.2? Make arguments of formula evaluable.**

To apply step 2.1, BAR must first reformulate Void in terms of Disjoint:

> **Goal:**
> **Rewrite (Void $p_0$ $s_0$) to match**
> **(Disjoint (SetOf ?x ?U ?P) (SetOf ?y ?U ?Q))**

BAR finds 5 plans to get from Void to Disjoint; the user accepts the top-ranked one:

**2.1.1! Unfold definition of Void:**
**(Void $p_0$ $s_0$) →**
**(Not (Exists c (Cards-in-hand $p_0$) (In-suit c $s_0$)))**

**2.1.2? Fold into instance of**
**Disjoint(S1,S2) = (Not (Exists x S1 (In x S2)))**

To do step 2.1.2, BAR must reformulate In-suit in terms of set membership:

> **Goal:**
> **Rewrite (In-suit c $s_0$) to match (In x S2)**

BAR finds 4 plans to get from In-suit to In, and the user accepts the top-ranked one:

**2.1.2.1! $P_c$ → (In c (SetOf y S $P_y$)), where c has type S**

This "jittering" rule [Fickas 80] rewrites an arbitrary predicate P on an object c in terms of set membership: c satisfies P if c belongs to the set of things that satisfy P. Here this rule produces the transformation

**(In-suit c $s_0$) → (In c (SetOf y (Cards) (In-suit y $s_0$)))**

This enables the previous plan to be completed:

**2.1.2! Fold into instance of Disjoint:**
**(Not (Exists c (Cards-in-hand $p_0$)**
**(In c (SetOf y (Cards) (In-suit y $s_0$)))))**
**→ (Disjoint**
**(Cards-in-hand $p_0$)**
**(SetOf y (Cards) (In-suit y $s_0$)))**

To match the left side of the disjoint subsets rule, the first argument must be rewritten:

> **Goal:**
> **Rewrite (Cards-in-hand $p_0$) to match (SetOf x ?U ?P)**

BAR finds one plan to get from Cards-in-hand to SetOf and applies it:

**2.1.3! Unfold definition:**
**(Cards-in-hand $p_0$) → (SetOf x (Cards) (Has $p_0$ x))**

The disjoint-subsets rule can now be applied:

**2.1! Use probability formula for disjoint subsets:**
**(High (Pr (Disjoint**
**(SetOf x (Cards) (Has $p_0$ x))**
**(SetOf y (Cards) (In-suit y $s_0$)))))**
**→ (High (Pr-disjoint-formula**
**(# (SetOf x (Cards) (Has $p_0$ x)))**
**(# (SetOf y (Cards) (In-suit y $s_0$)))**
**(# (Cards))))**

**2.2? Make arguments of formula evaluable.**

The resulting expression is evaluable except for the predicate Has, which is not computable. BAR finds 3 plans for evaluating the first argument of the formula, the user accepts the top-ranked one, and BAR carries it out:

**2.2.1! Fold into instance of computable function:**
**(# (SetOf x (Cards) (Has $p_0$ x))) →**
**(Number-cards-in-hand $p_0$)**

This completes the original plan:

**2! Expression is now evaluable:**
**(High (Pr-disjoint-formula**
**(Number-cards-in-hand $p_0$)**
**(# (SetOf y (Cards) (In-suit y $s_0$)))**
**(# (Cards))))**

BAR halts, having achieved its original goal of operationalizing the condition "player $p_0$ is void in suit $s_0$" by reformulating it as there being a high probability that player $p_0$'s hand is disjoint from suit $s_0$, based on the size of the hand. A more accurate estimate was derived in FOO by restricting the universe to the set of unplayed cards, but this refinement is omitted here for brevity.

BAR's generality is illustrated by the fact that it can operationalize (Void $p_0$ $s_0$) in very different ways by following different plans. Plan P2 exploits the assumption that player $p_0$ plays legally, and leads to the inference that a player who fails to follow suit must be void. The main subproblem consists of inferring that the condition (Void $p_0$ $s_0$) is implied by the axiom (Legal p (Card-played-by p)) when (Suit-led) = $s_0$ and (Not (In-suit (Card-played-by $p_0$) $s_0$)). Plan P3 is based on remembering past events, and leads to the inference that a player who was void earlier in the round must still be void now. The key subproblem is to prove that a player who is void remains void. This is done by showing that becoming unvoid would require obtaining a card, which cannot occur during the course of play.

These solutions can be combined to produce the solution "a player who failed to follow suit earlier is definitely void; otherwise, if few cards are left, the player is likely to be void." The order of composition is based on the fact that plan P2 leads to a correct semi-decision procedure whose usefulness is extended by P3, while P1 produces a total but approximate decision procedure. BAR lacks an explicit understanding of these factors, and does not combine solutions.

## 6. Improvements to the problem-solver

The construction of a general advice-taker remains a long-term goal posing some difficult problems not addressed here, notably integrating multiple pieces of advice. In addition, BAR inherits many limitations of FOO, such as a far-from-complete set of transformations. However, experience with BAR has exposed several more specific problems worth mentioning here.

First, BAR uses a very simple all-or-none model of operationality: either an expression can be evaluated or it cannot. This model ignores such factors as the cost of evaluation, the fact that some expressions can be evaluated sometimes but not always, and the semantic relationship between the original advice and its (perhaps heuristically) operationalized form.

Since BAR does not model the relative quality of alternative solutions, it cannot explicitly model the notion of refining a crude solution into an improved one. The refinement paradigm was evident in more than one example generated using FOO, especially in the synthesis of a heuristic search procedure by applying optimizing transformations to an initial generate-and-test search [Mostow 82]. A simple way to incorporate this into the means-ends analysis paradigm is to treat refinement somewhat like simplification: designate certain rules as refinement rules and automatically include them in the set of options whenever they appear applicable to the current expression (at the cost of increasing the branching factor). Of course the decision whether to actually apply such a rule depends on its relative costs and benefits in the case at hand.

To make operationalization totally automatic, BAR's problem-solver must be substantially improved. One problem is that BAR's depth-first search strategy is too sensitive to selecting the wrong plan, which causes exhaustive exploration of that branch of the search tree. A more cautious breadth-first strategy might avoid this pitfall, but implementing it would be complicated by the recursive nature of the problem-solver.

The current plan-rating scheme suffers from the horizon effect in that problems pushed down to a lower level are ignored. For example, plan P1 receives an unrealistically high rating because it ignores the problem of evaluating the arguments of Disjoint. The ratings could be improved by identifying obstacles to the evaluation of an expression and estimating the difficulty of eliminating each one. Identifying the obstacles seems straightforward, but it is not clear how to estimate their difficulty without actually solving them.

A fundamental problem with the planner is caused by using CAR as the abstraction function. This works fine on instances of highly specific functions, like Void, but loses too much information when applied to quantifiers like Exists, logical connectives like Not and And, and general functions like In. This results in the generation of silly plans. This problem might be ameliorated by designing a more discriminating abstraction function. Some such improvement is essential to constrain the generation of abstract plans; although the current branching factor of under 10 is a vast improvement over FOO, the length of the transformation sequences (sometimes over 100) implies that total automation will require an average branching factor very close to 1.

## 7. Conclusion

BAR was implemented to make explicit the goal structure left implicit in FOO, and in this it has largely succeeded. Although it does not totally automate the selection of what operators to apply, it has reduced the branching factor by a factor of $10^3$, without even counting the effect of plan scoring. It has been used to solve about half the examples done with FOO, and has clarified the improvements needed to handle some of the others: a more sophisticated control structure and model of operationality that make explicit the heuristic nature of operationalization.

In short, BAR should be viewed as a problem-solving model rather than as a practical automatic tool, and the broad scope of the operationalization problem makes this likely to remain the case for improved versions of BAR in the forseeable future. However, techniques developed in BAR may soon find practical use in the area of program transformation. An especially promising application is the problem of reformulating system components described in the Gist specification language [Feather 83] in terms of the data and operations available to their implementations [Mostow 83]. Even partial automation of this process could significantly enhance human productivity in developing software [Fickas 82].

## References


[Darlington 76] J. Darlington and R. M. Burstall, "A system which automatically improves programs," *Acta Informatica* 6, 1976, 41-60.

[Dietterich et al 82] T. G. Dietterich, Bob London, K. Clarkson, and G. Dromey, "Mostow's Operationalizer," in P. R. Cohen and E. A. Feigenbaum, Volume 3 (eds.), *Handbook of Artificial Intelligence*, Stanford Computer Science Department, Stanford, CA, 1982. In section on Learning and Inductive Inference, available as STAN-CS-82-913/HPP-82-10.

[Feather 83] M. S. Feather, Closed System Specifications, 1983. In preparation.

[Fickas 80] S. Fickas, "Automatic goal-directed program transformation," in *AAAI80*, pp. 68-70, American Association for Artificial Intelligence, Stanford University, 1980.

[Fickas 82] S. Fickas, *Automating the Transformational Development of Software*, Ph.D. thesis, University of California at Irvine, 1982.

[Hayes-Roth 81] F. Hayes-Roth, P. Klahr, and D. J. Mostow, "Advice taking and knowledge refinement: an iterative view of skill acquisition," in J. A. Anderson (ed.), *Cognitive Skills and their Acquisition*, pp. 231-253, Erlbaum, 1981.

[Kant & Newell 82] E. Kant and A. Newell, *Problem solving techniques for the design of algorithms*, Carnegie-Mellon University Computer Science Department, Technical Report CMU-CS-82-145, November 1982. To appear in *Information Processing and Management*.

[McCarthy 68] J. McCarthy, "The advice taker," in M. Minsky (ed.), *Semantic Information Processing*, pp. 403-410, MIT Press, Cambridge, MA, 1968.

[Mostow 79] D. J. Mostow and F. Hayes-Roth, "Operationalizing heuristics: some AI methods for assisting AI programming," in *IJCAI-5*, pp. 601-609, Tokyo, Japan, 1979.

[Mostow 81] D. J. Mostow, *Mechanical Transformation of Task Heuristics into Operational Procedures*, Ph.D. thesis, Carnegie-Mellon University, 1981. Technical Report CMU-CS-81-113.

[Mostow 82] D. J. Mostow, "Learning by being told: Machine transformation of advice into a heuristic search procedure," in J. G. Carbonell, R. S. Michalski, and T. M. Mitchell (eds.), *Machine Learning*, Palo Alto, CA: Tioga Publishing Company, 1982.

[Mostow 83] J. Mostow, "Operationalizing advice: a problem-solving model," in *Proceedings of the International Machine Learning Workshop*, University of Illinois, June 1983.

[Newell 60] A. Newell, J. Shaw, H. Simon, "Report on a general problem-solving program for a computer," in *Proceedings of the International Conference on Information Processing*, pp. 256-264, UNESCO, Paris, 1960.

[Newell 79] A. Newell, *Reasoning, problem solving and decision processes: the problem space as a fundamental category*, Carnegie-Mellon University Computer Science Department, Pittsburgh, PA, Technical Report, June 1979.

[Sacerdoti 74] E. D. Sacerdoti, "Planning in a hierarchy of abstraction spaces," *Artificial Intelligence* 5, 1974, 115-135.