Physical schemas for planning domain analysis

Robert St. Amant
Department of Computer Science
North Carolina State University
Box 7534
Raleigh, NC 27695-7534
stamant@csc.ncsu.edu

Abstract

Analysis of a planning domain has been shown to produce information that can greatly reduce the space of plans a controller must search through. Domain analysis is generally based on the logical inferences one can make given a set of planning operators and sometimes information about goals. An alternative heuristic approach is possible as well. We have developed a small set of physical schemas for classifying operators in planning domains. We show how they perform on some commonly available domains and describe how this information can help to explain the structure of a plan in physical, task-oriented terms.

Introduction

Planning algorithms have taken dramatic steps over the past few years. Modern planning algorithms such as GraphPlan and SAT-based algorithms can solve problems that are orders of magnitude more difficult than earlier algorithms (Weld 1999). Still, as might be expected, many large or difficult problems remain beyond the abilities of the current generation of planners, and may always do so. More interestingly, especially for researchers in collaborative or mixed-initiative planning, many problems that people find relatively easy to solve also remain difficult for planning algorithms.

McDermott's grid world (originally Manhattan world) is one well-known example (McDermott 1996; 1997). In a typical grid world problem, an agent must plan its way among locations on an n by n grid, each location containing a key but surrounded by locks. Solutions have the flavor, "Travel to location (x_1, y_1), entering from the north; get key `dk`; travel to the lock `dl` immediately east of location (x_2, y_2), unlock it and retrieve key `ck`..." Eventually the agent gains access to its goal location. McDermott's UNPOP planner takes between 15 minutes and almost an hour to solve such a problem, depending on its search strategy; an implementation of GRAPHPLAN is reportedly faster, but it is not clear by how much. In contrast, people find such problems almost trivially easy. Grid world is not alone in demanding what seems to an inordinate amount of search—even some blocks world problems are more difficult for planners than they initially appear.

A number of reasons are responsible for the performance differences between people and domain-independent planning algorithms on problems like these, including differences in search strategy and problem representation. This paper concentrates on the contribution of the "physical" heuristics we apply in solving problems. We have an enormous store of knowledge about how objects behave and interact in the real world, knowledge we can bring to bear in reducing or eliminating search. In blocks world problems, for example, we know that towers must be built from the bottom up. We know that placing one block on top of another makes the latter block inaccessible for further stacking or unstacking. In grid world problems, we know that moving back and forth between two locations will not get us closer to our goal, whatever that may be. We know that the path we take to a given location is often irrelevant, as long as we arrive at that location. This kind of knowledge is in one sense domain-dependent, in that it concerns assumptions about common types of object relationships and interactions. In another sense, however, this knowledge is extremely general: some researchers have argued that our experience with physical objects and environments provides the foundation for our understanding of more abstract concepts, via metaphor (Lakoff 1984).

The domains we encounter in planning research reflect our natural interest in physical domains. In this paper we show that some general types of physical operators occur across a variety of domains, and we can identify them not only with our external knowledge but also by their internal structure. If a planner can automatically impose a physical interpretation on the operators in a domain, two benefits are possible. First, it facilitates a more natural cooperation with human users in cooperative planning process. For example, instead of a planner stating that it is trying to satisfy a sequence of `(at ?p)` and `(adjacent ?p1 ?p2)` goals, it might state that its goal is movement from one location to another. Second, it provides a framework in which a planner can reason about goal relationships (Wilensky 1981). That is, a planner might use heuristic knowledge about physical interactions to guide its search. This is a relatively old idea, but deserves another look now that standardized planning languages are becoming more widespread.

The remainder of this paper is structured as follows. In the AFS section we describe the Abstract Force Simulator, an AI planning and simulation testbed that provided the initial context for our work with physical schemas. In the operator classification section we describe a number of templates that represent common ways of interacting with the physical world. These templates capture problem space representations of movement, ordering, accessibility, operator pairs, and other types of operators. We give a sample of their results for some of the UCPOP domains supplied with the PDDL specification software (Ghallab *et al.* 1998). In the plan interpretation section we describe how these templates can be used to generate descriptions of plans. The resulting descriptions do not provide major abstractions, but they do create a useful link between predicates and physical concepts. Our longer term goals for this work include a plan search controller based on our physical schemas. Unfortunately, this effort is at an early stage. We discuss ways in which physical schemas can potentially reduce search, but only in brief, general terms, and we do not present an implementation.

AFS

During the past year we have been working with the Abstract Force Simulator, a general-purpose simulation system that supports experimentation with interactive, distributed planning techniques and their relationship to physical processes (Atkin *et al.* 1998). AFS provides a physical domain for objects to interact in, based in general on Newtonian physics. Objects have mass, size, and shape; they may be solid or permeable; they move with variable friction over a flat surface; they apply force to one another, which causes mass reduction. Plan execution and monitoring is through HAC, a hierarchical planner embedded in AFS.

In AFS's Capture the Flag domain, two teams of agents move over a terrain, their travel constrained by mountains, water, and forests. Each team is responsible for defending a set of stationary flags, and successfully completes a scenario by destroying the members of the opposing team or capturing all of its flags. In this domain, as in all AFS domains, agents rely on a small set of primitive physical actions: they may `move` from one location to another and `apply-force` to other agents and objects such as flags. These actions can be specialized and combined in various ways to form higher level strategies for action, such as blocking a pass, encircling a flag, attacking an enemy in a group, and so forth.

In implementing a library of plans for our agents we found a surprising degree of simplicity and commonality at the lowest level of abstraction. Consider movement. In AFS, agents move over a continuous space by taking discrete steps, a location established at each simulation tick. In a conventional planning domain, movement is almost always implemented through operators that represent the location of the agent. Movement then consists of "retracting" the current location of the agent and "asserting" some new location, where retraction and assertion may be logic operations or may be implicit in a more procedural representation. We find that unless we adopt a drastically different representation for the physics of agent behavior, all kinds of movement involve this pattern. If we know that a given operator constitutes incremental movement over some space, then we have a natural heuristic for selecting its arguments: *if moving from A to B at some step j, don't move from B to A at step $j + 1$.* What is interesting about this heuristic is that the specific type and context of the movement usually don't matter (in AFS); the HAC planner can rely on a general-purpose path planner that implements the no-immediate-revisits policy and expect it to be reasonably successful.

Consider a slightly more abstract example, accessibility. In Capture The Flag problems, some areas of the terrain are inaccessible due to its division into mountainous and water regions. Other areas can be rendered temporarily inaccessible by sending agents to defend strategic passes, which act as cut points. The same relationship may arise if flags are defended sequentially, a team defending the flags nearest the front and falling back when it is overcome. In such cases we can think of the attacking team as acting on one object or area to make another accessible. The HAC planner can use this information in a straightforward way to make ordering decisions in a plan of attack.

It turns out that almost all activities in AFS can be decomposed into coordinated combinations of movement and the application of force, even fairly abstract notions such as forcing an opponent into a weak position, channeling an opponent's efforts, delaying, and feinting. The heuristics we use in selecting and understanding physical actions, based for example on the notion of accessibility, have a great deal of generality in the physical domains of AFS. Does this apparent generality extend to conventional domain-independent planning? We discuss this issue in the next two sections.

Classifying operators

A set of planning operators represents the physics of a domain (Ghallab *et al.* 1998). This physics is commonly much more abstract than that described for AFS. Nevertheless, some physical aspects are almost always retained: some objects have spatial extent (if only location), some objects are movable, some objects may contain other objects, an object may exist in specific relationships with other objects, and so forth. Such aspects, present to some extent in most planning domains, motivate our development of a relatively simple set of operator classifiers to analyze the structure of an operator in isolation or in comparison to other operators in a domain. The classifiers parse operators in PDDL format (Ghallab *et al.* 1998), handling conditionals but not quantification.

In the symbolic description of each classifier, we assume that each operator has a precondition and an effect, each of which is a conjunction of predicates, accessible through the functions $p(M)$ and $e(M)$. These functions have further, specialized versions: $p^+(M)$ and $e^+(M)$ give only the atomic predicates in the form, $p^-(M)$ and $e^-(M)$ the negated predicates. Predicates are Greek symbols whose predicate symbol is given by the function $n()$ and variable arguments by $a()$. For convenience, we use infix set functions and equality, and a unification function. Though we give a description in logical form for the classifiers, the implementation is via procedures specialized to each classifier.

Movement: A movement operator maintains a term representing an object's location. The operator represents movement by retracting the object's current location and asserting some new location. In some domains, the operator relies on a term that defines adjacency relationships between locations. In other, simpler domains, the operator may assume that all locations are adjacent to all others, and either asserts or assumes that the current location and new location are not identical. This kind of information is not always explicit but can often be automatically derived (as an implicative constraint (Gerevini & Schubert 1998)); our classifier does not do such processing. An example of a movement operator is given in Figure 1, with the relevant terms in boldface.

$$
\begin{aligned}
&\mathit{Movement}(M):\\
&\quad \exists(\phi,\psi) : \phi \in p(M) \wedge \neg\phi \in e(M) \wedge\\
&\qquad\qquad \psi \in e(M) \wedge\\
&\qquad\qquad n(\phi) = n(\psi) \wedge\\
&\qquad\qquad [\neg \mathit{unifies}(\phi,\psi) \vee\\
&\qquad\qquad\ \exists(\xi) : \xi \in p(M) \wedge a(\phi) \cup a(\psi) \subseteq a(\xi)]
\end{aligned}
$$

Ordering: An ordering operator maintains information about the relationship between objects, comparable to that provided by an adjacency relationship. An ordering relationship, however, may be asserted or retracted, unlike the usual treatment of adjacency. The operator represents ordering by testing arbitrary predicates on objects x and y in its precondition, and then asserting or retracting some single predicate that relates x and y in its effect. The intuition is that putting objects in a specific configuration involves testing whether specific conditions hold for each individually, and then imposing the ordering relationship on the objects together. See Figure 1 for an example.

$$
\begin{aligned}
&\mathit{Ordering}(M):\\
&\quad \exists(\phi,\psi,\xi) : \phi,\psi \in p(M) \wedge\\
&\qquad\qquad n(\phi) = n(\psi) \wedge\\
&\qquad\qquad [\neg \mathit{unifies}(\phi,\psi) \vee\\
&\qquad\qquad\ \exists(\xi) : \xi \in e(M) \wedge a(\phi) \cup a(\psi) \subseteq a(\xi)]
\end{aligned}
$$

Switches: Some operators are concerned with state changes represented by a single predicate. That is, these operators may do little else besides assert or retract a given term, the way a switch turns a light on or off. For a switch operator, a predicate is tested in the precondition and then negated in the effect, or vice versa, and there are no other applications of the predicate to other objects in the operator. The intuition behind this classification is that in some cases some binary property of a state may be necessary for the execution of one or more operators—essentially, the world enters a mode—and it can be important to know when the mode changes.

$$
\begin{aligned}
&\mathit{Switch}(M):\\
&\quad \exists(\phi) : \phi \in p(M) \wedge \neg\phi \in e(M) \wedge\\
&\qquad \neg[\exists(\psi) : \psi \in p(M) \cup e(M) \wedge \neg(\psi = \phi) \wedge\\
&\qquad\qquad n(\phi) = n(\psi)]
\end{aligned}
$$

Pairs: Some operators come in pairs, in which one generates some effect and the other undoes the effect. We see this in operators for loading and unloading a truck, for example. If every effect term in one operator is negated in the effect of another operator, and vice versa, we have a pair of do/undo operators, which are surprisingly common in planning domains and the real world.

$$
\begin{aligned}
&\mathit{Paired}(M,N):\\
&\quad \forall(\phi) : \phi \in e(M) \leftrightarrow \neg\phi \in e(N)
\end{aligned}
$$

Scoped: The execution of some operators is limited by their dependence on a predicate that can be negated by some operator but asserted by none. We test for this by testing an operator against all others in the domain to see whether any of its preconditions are negated by the effect of some operator but cannot then be asserted, or vice versa. The significance of this observation is that, in some sequences, these operators may only be able to be executed once, and then never again.

$$
\begin{aligned}
&\mathit{Scoped}(M):\\
&\quad [\exists(\phi) : \phi \in p^+(M) \wedge\\
&\qquad [\exists(N) : N \in \mathit{domain}(\phi) \wedge\\
&\qquad\qquad \exists(\psi) : \psi \in e^-(N) \wedge n(\phi) = n(\psi)] \wedge\\
&\qquad \neg[\exists(P) : P \in \mathit{domain}(\phi) \wedge\\
&\qquad\qquad \exists(\psi) : \psi \in e^+(P) \wedge n(\phi) = n(\psi)]] \vee\\
&\quad [\exists(\phi) : \phi \in p^-(M) \wedge\\
&\qquad [\exists(N) : N \in \mathit{domain}(\phi) \wedge\\
&\qquad\qquad \exists(\psi) : \psi \in e^+(N) \wedge n(\phi) = n(\psi)] \wedge\\
&\qquad \neg[\exists(P) : P \in \mathit{domain}(\phi) \wedge\\
&\qquad\qquad \exists(\psi) : \psi \in e^-(P) \wedge n(\phi) = n(\psi)]]
\end{aligned}
$$

Unique and top-level effects: Some operators produce effects that are not used by other operators; these are unique effect operators. A variant simply tests whether an operator produces an effect that is part of the goal of a specific problem (if we make a problem available to the classifier). Intuitively, these operators should have some prominence, in comparison to operators that only satisfy the preconditions of other operators.

$$
\begin{aligned}
&\mathit{Unique}(M):\\
&\quad \exists(\phi) : \phi \in e(M) \wedge\\
&\qquad\quad \neg[\exists(N) : N \in \mathit{domain}(\phi)\ \wedge\\
&\qquad\qquad\qquad \neg[\exists(\psi) : \psi \in p(N) \wedge n(\phi) = n(\psi)]]
\end{aligned}
$$

These heuristic classifiers capture much of what we would consider physical activity in a domain: moving between locations, putting objects in specific ordered relationships, establishing modes and shifting between them. (Other classifiers for containment, carrying, and comparable activities remain to be built.) Of course, the weakness of all these classifiers is obvious. They detect very specific patterns in single operators and combinations of operators, and will fail for simple variations. We can imagine movement operators, for example, with preconditions that impose arbitrarily complex constraints on movement from one location to another. In practice, fortunately, the classifiers do tend to identify operators correctly. Figure 2 shows how they classify operators in a number of domains developed for UCPOP. Though we cannot argue for the generality of these classifiers based on this small sample, we find the degree of simple detectable structure suggestive.

For example, movement is identified with `mov-b` (briefcase-world), `go` (grid-world), `push-box` (monkey), `cross` and `drive` (road-operators), and `move` (robot), among others. We find no examples of movement that are not represented in this way. Similarly, `stack` and `puton` operators are classified as ordering operators in all the blocks world formulations we have tested. A number of paired operators are found: `put-in/take-out` (briefcase-world), `board/debark` (ferry), `screw/unscrew` (fridge), and many others. As for scoped operators, we find that in the monkey domain, once the agent has climbed onto the box (producing `(:not (on-floor))`), it is unable to continue with `go-to` or `push-box`. Other examples can be seen in Figure 1.

Classification is not perfect—its requirements are too lenient. In some cases, results are questionable or incorrect; for example, `open-door` (strips-world) and `move-disk` (hanoi) are classified as movement operators, among other possibilities. Examining a specific problem can help. For example, because the number of predicates potentially referenced by `open-door` is so small, it would not help much to treat it as a movement operator. The same is true for a number of operators that are nominally ordering operators. A priori precedence in interpreting the various classifications associated with an operator can also help, but will require more study to be reliable.

Interpreting plans

Plan search control is a natural direction for deciding whether these classifiers are useful: do they improve the performance of a planner? Unfortunately, we have only begun to examine the issues in this area; we have experimented with building a search controller for a specialized physical planner, but it is not complete. [1]

Instead, we turn to a weaker form of evaluation, but one that holds significant interest for researchers in collaborative planning. A collaborative planning system interacts with a human user, exchanging information about the planning process in order to construct a plan that may be beyond the ability of the human or the planning system acting alone. In this kind of interaction a key capability for a planner is the presentation of its results, partial and complete plans, in a comprehensible form. For many collaborative planning systems, even those based on domain-independent planners, the interaction must take place in the environment of a domain-dependent user interface; TRIPS is one familiar example (Ferguson & Allen 1998). Other approaches that show the structure of a plan under construction, or ask for assistance in deciding between alternatives (*Should variable ?TRUCK be bound to TRUCK-12 or TRUCK-13?*) have drawbacks in the amount of information they expect users to absorb and the level of abstraction at which the interaction takes place.

One of the ways people use their physical understanding of a situation is to organize relevant information. Imagine a long, staged sequence of movement and ordering operators, for example, that I can summarize by aggregating into *gathering objects*, *carrying them*, and *distributing them again*—the container/conduit metaphor (Lakoff 1984). The added structure, imposed by a physical interpretation of the operators, can make plans easier to understand. We have developed a small number of methods for automatically generating descriptions of totally ordered plans, using the classifications of their operators.

Aggregated movement: Movement is a natural candidate for aggregation of similar operators that appear in sequence. If we see several movement operators of the same type, we can aggregate them by simply referring to their endpoints, as shown in Figure 3 for the road-operators domain.

Accessibility: In many physical tasks, we need to manipulate an object that is not accessible, perhaps in a container or behind or under other objects. In the latter case, we move the blocking objects out of the way and see if it makes the goal object accessible. If not, we recurse. Ordering operators impose a relationship of recursive accessibility on a set of objects. Figure 3 shows how ordering operators can be interpreted, given information about the goal in a blocks world problem. The first several steps are interpreted as simply making other objects accessible.

Prepare/do/finish structure: Many real-world tasks involve setting up a situation for a specific set of actions, executing them, and then cleaning up afterwards—think of the way we prepare a meal, or

[1] We expect that the heuristics described in this section could be built into a best-first search controller for GRAPHPLAN, but we have not pursued this idea.

repair a sink, or paint a room. Paired operators are natural candidates for components in this three-part structure. For example, in the flat tire domain, the operators `Remove-wheel` and `Put-on-wheel` surround the main activity of replacing the tire. There are two heuristics here. First, singletons from paired operators are aggregated as a preparation cluster, up to the point where an unpaired operator appears or a match appears that signals a finishing cluster. Second, these clusters are interpreted as leading to a main action, which is often flagged by the occurrence of a unique effect or top level operator. Figure 3 shows a common nested structure for a plan in the fridge domain.

For these examples, we have chosen the clearest and most representative cases. Nevertheless, because of the weakness of operator classification, these physical interpretations are often flawed. To give one example, in the flat tire domain `undo` is not correctly paired with `do-up`, so that the aggregation of steps into a single "prepare" step is inappropriately broken up.

Even in these cases, however, it is possible to see how these kinds of physical or task-related interpretations of operators can contribute to reduced plan search. If an accessibility relationship between objects with respect to specific operators can be derived automatically, not only through interpretation of an existing plan, but from analysis of an initial state (which we have not yet succeeded in doing), a planner could apply the heuristic, *If object A is inaccessible due to object B, then make A accessible before satisfying a top level goal for B.* In the case of aggregated pairs of operators in a prepare/do/finish structure, we observe that in all the real world domains we can think of, there are only three possibilities for action ordering: first, the ordering of preparation steps must be the same as finishing steps; second, the ordering must be reversed; third, the ordering is irrelevant. Another heuristic arising from this structure involves expecting a unique effect or top level operator to close out a preparation sequence; initiating a finish action before such a step can be a mistake.

We emphasize that, as with operator classification, these are weak methods. Nevertheless they take steps in what we consider a useful direction for domain-independent planning: making their results more comprehensible via connections to a physical understanding of the world.

Related work

This work was inspired in part by McDermott's UNPOP system (1996; 1997), which applies means-ends analysis in a novel way to the planning problem. UNPOP, like GPS and Prodigy, searches for plans through situation space. Unlike these planners, instead of maintaining state information that includes a structure of goals remaining to be satisfied, UNPOP maintains only plan prefixes as search states and at each step constructs a chain leading backwards to relevant, feasible actions. In a discussion of the performance of partial order planners on grid world, McDermott notes that without a situation-based state representation, a planner cannot tell that moving back and forth between two locations makes no progress. In UNPOP, such movement heuristics are a natural outcome of the situation space search.

Others have pursued domain analysis to improve the performance of specific planners. Gerevini and Schubert (1998), for example, propose techniques for automatically speeding up SAT-based planners. The first set of techniques infers state constraints from the structure of planning operators and the initial state. For example, in some blocks world domains one can infer that when one block is on top of another block, then the lower block is not clear (an implicative constraint), or that no block can be on more than one block at any time (a single-valuedness constraint). The second set of techniques infers predicate domains, co-occurrences of predicate arguments. Weld briefly discusses other approaches to domain analysis (Weld 1999). Our work has similar aims, and could benefit from these techniques. One difference is that instead of examining heuristics that classes of planners find helpful, we are interested in general-purpose heuristic knowledge about common tasks that can impose external structure on plans.

References

Atkin, M.; Westbrook, D. L.; Cohen, P. R.; and Jorstad, G. D. 1998. Afs and hac: Domain-general agent simulation and control. In *AAAI-98 Workshop on Software Tools for Developing Agents*, 89–95.

Ferguson, G., and Allen, J. 1998. TRIPS: An intelligent integrated problem-solving assistant. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence.*

Gerevini, A., and Schubert, L. 1998. Inferring state constraints for domain-independent planning. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, 905–912.

Ghallab, M.; Howe, A.; Knoblock, C.; McDermott, D.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. *PDDL—The Planning Domain Definition Language.* AIPS-98 Planning Committee.

Lakoff, G. 1984. *Women, Fire, and Dangerous Things.* University of Chicago Press.

McDermott, D. 1996. A heuristic estimator for means-ends analysis in planning. In *Proceedings of the Third International Conference on Artificial Intelligence Planning Systems.*

McDermott, D. 1997. Using regression-match graphs to control search in planning. Under review.

Weld, D. S. 1999. Recent advances in ai planning. *AI Magazine.* To appear.

Wilensky, R. 1981. Meta-planning: representing and using knowledge about planning in problem solving and natural language understanding. *Cognitive Science* 5:197–233.

```
;;; Movement operator (grid world domain)
(:action Go
         :parameters (?x ?y ?nx ?ny)
         :precondition (:and (at-robot ?x ?y)
                             (path ?x ?y ?nx ?ny)
                             (:not (:exists (key ?k)
                                            (lock ?nx ?ny ?k))))
         :effect (:and (:not (at-robot ?x ?y))
                       (at-robot ?nx ?ny)))

;;; Ordering operator (Prodigy blocks world domain)
(:action Stack
         :parameters (?sob ?sunderob)
         :precondition (:and (holding ?sob) (clear ?sunderob))
         :effect (:and (:not (holding ?sob))
                       (:not (clear ?sunderob))
                       (clear ?sob)
                       (arm-empty)
                       (on ?sob ?sunderob)))

;;; Paired operators (flat-tire domain)
(:action Remove-wheel
         :parameters ((wheel ?x) (hub ?y))
         :precondition (:and (:neq ?x ?y) (:not (on-ground ?y))
                             (on ?x ?y) (unfastened ?y))
         :effect (:and (have ?x) (free ?y) (:not (on ?x ?y))))

(:action Put-on-wheel
         :parameters ((wheel ?x) (hub ?y))
         :precondition (:and (:neq ?x ?y) (have ?x) (free ?y) (unfastened ?y)
                             (:not (on-ground ?y)))
         :effect (:and (on ?x ?y) (:not (have ?x)) (:not (free ?y))))

;;; Switch operator (fridge domain)
(:action screw
         :parameters ((screw ?x) (backplane ?y))
         :precondition (:and (:not (screwed ?X)) (holds ?x ?y))
         :effect (screwed ?X))

;;; Scoped operator (flat-tire domain)
(:action inflate
         :parameters ((wheel ?x))
         :precondition (:and (have pump) (:not (inflated ?x)) (intact ?x))
         :effect (inflated ?x))

;;; Unique effect operator (monkey domain)
(:action GRAB-BANANAS
         :parameters (?y)
         :precondition (:and (hasknife) (at bananas ?y) (onbox ?y))
         :effect (hasbananas))
```

Figure 1: Some operators and their classifications

Domain: Operator	**Matching schemas**
blocks-world: *puton*	ordering(on), unique-effect(not clear) (not on)
briefcase-world: *mov-b*	movement(at), unique-effect(not at)
briefcase-world: *put-in*	paired(*take-out*)
briefcase-world: *take-out*	paired(*put-in*), unique-effect(not in)
ferry: *board*	switch(empty-ferry at), paired(*debark*), unique-effect(not empty-ferry) (not at)
ferry: *debark*	ordering(at), switch(on), paired(*board*), unique-effect(not on) (ferry), top-level(at)
ferry: *sail*	unique-effect(not at-ferry)
fridge: *attach-backplane*	switch(in-place), paired(*remove-backplane*)
fridge: *change-compressor*	ordering(covers), unique-effect(not covers), top-level(attached)
fridge: *remove-backplane*	switch(in-place), paired(*attach-backplane*)
fridge: *screw*	switch(screwed), paired(*unscrew*)
fridge: *start-fridge*	switch(fridge-on), paired(*stop-fridge*)
fridge: *stop-fridge*	switch(fridge-on), paired(*start-fridge*)
fridge: *unscrew*	switch(screwed), paired(*screw*)
hanoi: *move-disk*	movement(on), ordering(on), unique-effect(not clear) (not on)
ho-world: *fix*	unique-effect(not holey-walls)
ho-world: *turn-faucet*	unique-effectholey-walls (not water)
flat-tire: *close*	switch(open), paired(*open*), top-level(not open)
flat-tire: *cuss*	unique-effect(not annoyed)
flat-tire: *do-up*	ordering(loose), switch(unfastened)
flat-tire: *fetch*	switch(in), paired(*put-away*), unique-effect(not in)
flat-tire: *inflate*	switch(inflated), scoped(not inflated), unique-effect(inflated), top-level(inflated)
flat-tire: *loosen*	ordering(loose), switch(tight), paired(*tighten*), unique-effect(not tight)
flat-tire: *open*	switch(open), paired(*close*)
flat-tire: *put-away*	ordering(in), switch(have), paired(*fetch*), top-level(in)
flat-tire: *put-on-wheel*	ordering(on), switch(free have), paired(*remove-wheel*), unique-effect(not free), top-level(on)
flat-tire: *remove-wheel*	switch(on), paired(*put-on-wheel*)
flat-tire: *tighten*	ordering(tight), switch(loose), paired(*loosen*), top-level(tight)
flat-tire: *undo*	switch(unfastened)
mcd-grid-world2: *go*	movement(at-robot), unique-effect(not at-robot)
mcd-grid-world2: *pickup*	switch(at), scoped(at), unique-effect(not at)
mcd-grid-world: *go*	movement(at-robot)
mcd-grid-world: *open*	movement(at-robot)
mcd-grid-world: *pickup*	switch(at), scoped(at), unique-effect(not at)
molgen: *cleave*	switch(cleavable)
molgen: *digest*	switch(hair-pin), unique-effect(not hair-pin)
molgen: *ligate*	ordering(contains), unique-effect(not cleaved)
molgen: *polymerize*	switch(single-strand), unique-effect(not single-strand)
molgen: *screen*	unique-effect(pure)
molgen: *separate*	switch(connected-cdna-mrna), unique-effect(not connected-cdna-mrna)
molgen: *transform*	ordering(contains), switch(cleavable)
monkey: *climb*	unique-effect(not on-floor)
monkey: *getwater*	unique-effect(haswater)
monkey: *go-to*	scoped(on-floor)
monkey: *grab-bananas*	unique-effect(hasbananas)
monkey: *push-box*	movement(at), scoped(on-floor)
office-world: *move*	movement(at), ordering(at), unique-effect(not at)
office-world: *print-check-for*	ordering(written-for), unique-effect(written-for check)
office-world: *put-in*	ordering(in), paired(*take-out*)
office-world: *take-out*	switch(in), paired(*put-in*), unique-effect(not in)
prodigy-bw: *pick-up*	switch(arm-empty clear on-table), unique-effect(not on-table)
prodigy-bw: *put-down*	switch(holding), top-level(on-table)
prodigy-bw: *stack*	ordering(on), switch(holding), paired(*unstack*), top-level(on)
prodigy-bw: *unstack*	switch(on arm-empty), paired(*stack*), unique-effect(not on)
road-operators: *cross*	movement(at), ordering(at)
road-operators: *drive*	movement(at), ordering(at)
robot: *drop*	switch(grasping), paired(*pickup*), unique-effect(not grasping)
robot: *move*	movement(at), ordering(at), unique-effect(not at)
robot: *pickup*	switch(empty-handed), paired(*drop*), unique-effect(not empty-handed)
strips-world: *close-door*	movement(statis), paired(*open-door*)
strips-world: *go-thru-door*	movement(in-room), switch(next-to)
strips-world: *open-door*	movement(statis), paired(*close-door*)
strips-world: *push-box*	ordering(next-to)
strips-world: *push-thru-door*	movement(in-room), ordering(in-room)
strips-world: *push-to-door*	ordering(next-to)
strips-world: *push-to-loc*	ordering(at)

Figure 2: Classified operators in several domains

```
?  (sample-plan 'road-operators)
((CROSS MARK A D)
 (CROSS JACK A D)
 (DRIVE MARK D E)
 (DRIVE JACK D E)
 (DRIVE MARK E F)
 (DRIVE JACK E F)
 (DRIVE MARK F G)
 (DRIVE JACK F G))

?  (interpret-plan (sample-plan 'road-operators))
;;; Movement CROSS: old location status (AT MARK A); now (AT MARK D).
;;; Movement CROSS: old location status (AT JACK A); now (AT JACK D).
;;; Movement DRIVE: old location status (AT MARK D); now (AT MARK G).
;;; Movement DRIVE: old location status (AT JACK D); now (AT JACK G).


?  (sample-plan 'blocks-world-domain)
((PUTON C TABLE D)
 (PUTON D TABLE E)
 (PUTON E TABLE F)
 (PUTON F A G)
 (PUTON C D TABLE)
 (PUTON B C TABLE))

?  (interpret-plan (sample-plan 'blocks-world-domain) (sample-goal 'blocks-world-domain))
;;; Accessibility goal:(ON F A). PUTON:(C TABLE) status now "ON."
;;; Accessibility goal:(ON F A). PUTON:(D TABLE) status now "ON."
;;; Accessibility goal:(ON F A). PUTON:(E TABLE) status now "ON."
;;; Ordering PUTON:(F A) status now "ON."
;;; Ordering PUTON:(C D) status now "ON."
;;; Ordering PUTON:(B C) status now "ON."


?  (sample-plan 'fridge-domain)
((UNSCREW S4 B1) (UNSCREW S3 B1) (UNSCREW S2 B1) (UNSCREW S1 B1)
 (STOP-FRIDGE F1) (REMOVE-BACKPLANE B1 F1 S1 S2 S3 S4)
 (CHANGE-COMPRESSOR C1 C2 B1)
 (ATTACH-BACKPLANE B1 F1 S1 S2 S3 S4)
 (SCREW S4 B1) (SCREW S3 B1) (SCREW S2 B1) (SCREW S1 B1)
 (START-FRIDGE F1))

?  (interpret-plan (sample-plan 'fridge-domain))
;;; Set up with UNSCREW:S1, S2, S3, S4 status now not "SCREWED."
;;; Set up with STOP-FRIDGE:F1 status now not "FRIDGE-ON."
;;; Set up with REMOVE-BACKPLANE:B1 status now not "IN-PLACE."
;;; Main action CHANGE-COMPRESSOR:(C1 C2 B1)
;;;    C2 status now "ATTACHED."
;;; Finish with ATTACH-BACKPLANE:B1 status now "IN-PLACE."
;;; Finish with SCREW:S1, S2, S3, S4 status now "SCREWED."
;;; Finish with START-FRIDGE:NIL status now "FRIDGE-ON."
```

Figure 3: Some plan interpretations