Automatically Abstracting the Effects of Operators

Eugene Fink
Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada N2L3G1
efink@violet.waterloo.edu

Qiang Yang
Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada N2L3G1
qyang@logos.waterloo.edu

Abstract

The use of abstraction in problem solving is an effective approach to reducing search, but finding good abstractions is a difficult problem. The first algorithm that completely automates the generation of abstraction hierarchies is Knoblock's ALPINE, but this algorithm is only able to automatically abstract the *preconditions* of operators. In this paper we present an algorithm that automatically abstracts not only the preconditions but also the *effects* of operators, and produces finer-grained abstraction hierarchies than ALPINE. The same algorithm also formalizes and selects the *primary effects* of operators, which is thus useful even for planning without abstraction. We present a theorem that describes the necessary and sufficient conditions for a planner to be complete, when guided by primary effects.

1 INTRODUCTION

Recently, there has been an increasing amount of interest in formalizing abstraction and abstract problem-solving. Much work has stemmed from Sacerdoti's ABSTRIPS system [Sacerdoti, 1974], which builds an abstraction hierarchy by systematically eliminating preconditions of operators. Given a problem space and a hierarchy of abstractions, a hierarchical problem-solver first solves a problem in an abstract space, and then refines it in successively more detailed spaces. Abstraction often reduces the complexity of search by dividing up a problem into smaller subproblems.

A notable achievement in recent research is Knoblock's ALPINE system [Knoblock, 1991], which completely automates the formation of abstraction hierarchies. The hierarchies that ALPINE constructs satisfy the *ordered property* [Knoblock *et al.*, 1991], which states that while refining an abstract plan on a concrete level, no abstract-level predicate will be changed. To a large extent, the property is successful in formalizing an important intuition behind the use of abstraction: one wants to preserve the abstract-plan structure while adding concrete-level operators. Experiments [Knoblock, 1991] have demonstrated that in many problem domains, ALPINE gains an exponential amount of savings in planning time.

The ability for a problem-solver to reduce search relies heavily on the quality of an abstraction hierarchy it generates. A problem with ALPINE is that it often generates a hierarchy with too few levels. In some cases, the entire hierarchy collapses into a single level. The "collapsing-problem" of ALPINE makes it incapable of handling many real-life domains.

In this paper, we present a new algorithm for generating abstraction hierarchies that satisfy the ordered property. The new algorithm addresses the collapsing problem of ALPINE by providing more levels of hierarchies than ALPINE does. It often succeeds in cases where ALPINE fails to create a multi-level hierarchy. The intuition behind our success is due to the fact that our algorithm automatically abstracts not only preconditions but also the *effects* of operators. On the other hand, ALPINE only abstracts the *preconditions* of operators automatically, while its abstraction of effects depends critically on a set of user-provided primary-effects.

The paper starts by reviewing the terminology used in the planning literature and a brief discussion of the ALPINE system. Then we provide a theory for constructing abstraction hierarchies, and demonstrate the application of the described method via theorems and examples. Finally, we will discuss the advantages and possible shortcomings of our approach, and methods for fixing the shortcomings.

2 ABSTRACTION IN PLANNING: A REVIEW

We follow the terminology used in [Knoblock *et al.*, 1991]. A *planning domain* consists of a set of literals and a set of operators. Each *operator* α is defined by a set of *precondition literals* $Pre(\alpha)$ and *effect literals* $Eff(\alpha)$.

A *state* of the world is a set of literals. Applying an operator α produces a new state, where all literals from $Eff(\alpha)$ hold, and all literals that do not conflict with $Eff(\alpha)$ are left unchanged. For example, suppose p_1 and p_2 are some atomic statements in a problem domain. The corresponding literals are p_1, p_2, $\neg p_1$, and $\neg p_2$. Let $Eff(\alpha) = \{\neg p_1\}$. Then applying α to a state $S = \{p_1, p_2\}$ produces a new state $S = \{\neg p_1, p_2\}$.

A *plan* $\Pi = (\alpha_1, \ldots, \alpha_n)$ is a sequence of operators, which can be applied to a state by executing each operator in order. An *initial state* S_0 is a state of the world before executing plan, and S_i is the state achieved by executing first i operators of Π. A plan $\Pi = (\alpha_1, \ldots, \alpha_n)$ is *legal* relative to an initial state if the preconditions of each operator are satisfied in the state in which the operator is applied, i.e. $(\forall i \in [1 \ldots n])\ Pre(\alpha_i) \subseteq S_{i-1}$.

A *planning problem* is a pair (S_0, S_g), where S_0 is an initial state, and S_g is a *goal state.* A plan Π is *correct* relative to the planning problem (S_0, S_g) if Π is legal relative to S_0 and the goal is satisfied in the final state: $S_g \subseteq S_n$.

To build an abstraction hierarchy, one can associate some natural number, called *criticality*, with every literal in the problem domain. While solving problem at *the i-th level of abstraction*, we ignore all literals with criticality less than i. Abstract planning is usually done in a top-down manner: first we find an abstract solution at the highest level of abstraction. Then we refine it to account for successive levels of details. The process of transforming a correct plan Π from level i to a correct plan at a lower level is called *refinement.* Given a correct plan Π at level i of abstraction, the refinement process can be briefly described as follows:

1. Add the level $i-1$ literals into both the preconditions and effects of operators, and initial and goal states in Π
2. Add operators that achieve these new literals.

This concept of refinement captures the intuition behind plan refinements not only in ABSTRIPS, but also in the task-network based planning systems, such as SIPE [Wilkins, 1984] and NONLIN [Tate, 1977].

For a given set of operators, different criticality assignments may result in different behavior in problem solving. We would like an abstraction hierarchy to help reduce the complexity of planning. *Ordered hierarchies*, introduced in [Knoblock *et al.*, 1991], are aimed at achieving this goal. Informally, an abstraction hierarchy is an ordered hierarchy if every refinement of an abstract plan leaves the abstract plan structurally unchanged. Experiments have shown that the ordered hierarchies often increase the efficiency of planning algorithms exponentially [Knoblock, 1991]. ALPINE has been implemented by Knoblock to automatically generate ordered hierarchies that conform with human intuition behind "good" hierarchies. Hierarchies generated by ALPINE are based on the following syntactic restriction:

Restriction 1 (Ordered restriction)
Let O be the set of operators in a domain. $\forall \alpha \in O$, $\forall e_1, e_2 \in \mathrm{Eff}(\alpha)$, *and* $\forall p \in \mathrm{Pre}(\alpha)$ *such that* p *is achieved by some operator,*
(1) $\mathrm{crit}(e_1) = \mathrm{crit}(e_2)$, *and*
(2) $\mathrm{crit}(e_1) \geq \mathrm{crit}(p)$.

Intuitively, all effects of an operator have the same criticality, and their criticality is at least as great as the criticalities of operator's preconditions (except the preconditions that cannot be achieved by any operator).

Consider the Tower of Hanoi domain with 3 pegs and 3 disks. Let the three pegs be P_1, P_2, and P_3, and let the disks be $Large$, $Medium$ and $Small$. We can represent the locations of the disks using literals of the form *OnLarge*(x), *OnMedium*(x), and *OnSmall*(x). The operators for moving disks can be represented as shown in Table 1. Observe that the literals of the form *IsPeg*(x) cannot be achieved by any operator, while all other literals may be achieved by some operator.

We may assign criticality 3 to each *IsPeg* literal, criticality 2 to each *OnLarge* literal, criticality 1 to each *OnMedium* literal, and criticality 0 to each *OnSmall* literal. It is not hard to verify that this assignment satisfies the Ordered Restriction.

3 A MOTIVATING EXAMPLE

Suppose that in the Tower of Hanoi example, we add the operators that can move *two disks* at a time, as long as both disks are on the same peg, and there are

Table 1: Operators For the Tower Of Hanoi

Preconditions	*Effects*
MoveL(x,y)	
IsPeg(x), IsPeg(y), ¬OnSmall(x), ¬OnSmall(y), ¬OnMedium(x), ¬OnMedium(y), OnLarge(x)	¬OnLarge(x), OnLarge(y)
MoveM(x,y)	
IsPeg(x), IsPeg(y), ¬OnSmall(x), ¬OnSmall(y), OnMedium(x)	¬OnMedium(x), OnMedium(y)
MoveS(x,y)	
IsPeg(x), IsPeg(y), OnSmall(x)	¬OnSmall(x), OnSmall(y)

Table 2: New Operators For the Tower Of Hanoi

Preconditions	*Effects*
MoveLM(x,y)	
IsPeg(x), IsPeg(y), ¬OnSmall(x), ¬OnSmall(y), OnLarge(x), OnMedium(x)	¬OnLarge(x), OnLarge(y), ¬OnMedium(x), OnMedium(y)
MoveMS(x,y)	
IsPeg(x), IsPeg(y), OnSmall(x), OnMedium(x)	¬OnMedium(x), OnMedium(y), ¬OnSmall(x), OnSmall(y)
MoveLS(x,y)	
IsPeg(x), IsPeg(y), OnLarge(x), OnSmall(x), ¬OnMedium(x), ¬OnMedium(y)	¬OnSmall(x), OnSmall(y), ¬OnLarge(x), OnLarge(y)

no disks between them. The extra operators are listed in Table 2.

ALPINE fails to generate a hierarchy for this new domain: all literals collapse into a single level of abstraction. To see this, observe that according to Statement 1 of the Ordered Restriction, all effects of operators must have the same criticality. The operator *MoveLM* has effects *OnLarge* and *OnMedium*, and therefore $crit(OnLarge) = crit(OnMedium)$. Similarly, for effects of the operator *MoveMS* we have $crit(OnMedium) = crit(OnSmall)$.

So the only criticality assignment satisfying the Ordered Restriction is

$$crit(OnLarge) = crit(OnMedium) = crit(OnSmall).[1]$$

Notice that even with the operators for moving two disks, intuitively it is still true that moving the large disk is more difficult than moving the small one. Thus, intuitively one should still consider the movement of a large disk at an abstract level. This example shows a shortcoming of ALPINE: the addition of a few new operators may collapse an abstraction hierarchy into a single level, even though intuition tells us that the abstraction hierarchy should stay intact.

The purpose of this paper is to remove this deficiency of ALPINE. In the sections below, we achieve this by presenting a new algorithm that constructs abstraction hierarchies and still preserves the ordered monotonicity property.

4 ORDERED HIERARCHIES WITH PRIMARY EFFECTS

A key point to observe in the above example is that, if we want to move the small disk alone, we do not use the operator *MoveLS* or *MoveMS*. It is more natural to move the small disk with the operator *MoveS*. Similarly, we use the *MoveLM* operator if we want to move either the large disk[2], or the large and medium disks together. But we do not use *MoveLM* to move the medium disk alone. In other words, an operator is used for the sake of its *primary effects*. In the Tower of Hanoi example, we can envision $OnLarge(y)$ as the primary effect of the operator $MoveLS(x, y)$, and $OnSmall(y)$ as its *side effect*. The set of primary effects of the Tower of Hanoi operators are listed in Table 3.

As another example, suppose you are going to a computer shop to buy diskettes. The primary effect of this action is obtaining diskettes — this is your main goal. Side effects are spending $20, having a pleasant walk on a sunny day, wearing your shoes, and so on.

The purpose of recognizing primary effects is to reduce the complexity of a problem-solving process by reducing the branching factor of search space. When achieving some literal, a problem-solver needs to consider only operators whose primary effects contain this literal. If a problem-solver uses operators only for the sake of their primary effects, it is said to be *primary-effect restricted*. ALPINE is primary-effect restricted, but the primary effects of operators have to be provided by the user. Therefore, an extension of ALPINE would be to consider methods for automatically finding the primary effects that facilitate abstraction.

In the next section we present an algorithm for *automatically* finding the primary effects of operators. But for now, we assume that the primary effects of the operators in a domain have been found. The set of primary effects of an operator α is denoted by *Prim-Eff*(α). We now consider how to construct a finer-grained ordered abstraction hierarchy based on the primary effects. Consider the following modified ordered restriction:

Restriction 2 *Let O be the set of operators in a domain.* $\forall \alpha \in O$, $\forall e \in \text{Eff}(\alpha)$, $\forall e_1, e_2 \in \text{Prim-Eff}(\alpha)$, *and* $\forall p \in \text{Pre}(\alpha)$ *such that p is achieved by some operator,*
(1) $\text{crit}(e_1) = \text{crit}(e_2)$,
(2) $\text{crit}(e_1) \geq \text{crit}(e)$, *and*
(3) $\text{crit}(e_1) \geq \text{crit}(p)$.

This restriction formalizes the syntactic conditions behind the algorithm used by ALPINE ([Knoblock, 1991], page 83). For a primary-effect restricted problem-solver, Restriction 2 provides a sufficient condition to guarantee a hierarchy to be ordered:

Theorem 1 *If a planner is primary-effect restricted, then every abstraction hierarchy satisfying Restriction 2 is ordered.*

That is, if an abstraction hierarchy satisfies Restriction 2, then no new operators in the refinement of an abstract plan achieves an abstract literal, as long as the problem-solver is primary-effect restricted. To see that the theorem holds, consider the achievement of a precondition literal l at level $(i - 1)$ during the re-

[1] One may easily verify that a hierarchy collapses into a single level even if we use *problem-specific* ordered restrictions [Knoblock, 1991].

[2] The reason to use *MoveLM* when we need to move only the large disk is that the medium disk may be above the large one, in which case we cannot use *MoveL* without removing the medium disk first.

Table 3: Primary Effects Of Operators In the Extended Tower Of Hanoi.

Operators	Primary Effects	Operators	Primary Effects
MoveL(x,y)	OnLarge(y)	*MoveLM*(x,y)	OnLarge(y)
MoveM(x,y)	OnMedium(y)	*MoveMS*(x,y)	OnMedium(y)
MoveS(x,y)	OnSmall(y)	*MoveLS*(x,y)	OnLarge(y)

finement of an abstract plan. Suppose an operator α is selected for achieving l. Since our problem-solver is primary-effect restricted, l is a primary effect of α. The criticalities of all other effects of α are not greater than $(i-1)$, and therefore no higher-level literals are achieved by α. For a more formal proof, see [Fink, 1992]. It is easy to create an algorithm that automatically generates an abstraction hierarchy based on Restriction 2.

5 AUTOMATICALLY SELECTING PRIMARY EFFECTS

The construction of an ordered abstraction hierarchy for a primary-effect restricted problem-solver is based on a definition of primary effects for the set of operators in a domain. For each planning domain, there are different ways to define primary effects, which can be grouped into the following three categories:

1. **All Effects Are Primary Effects.** This option is implicitly used in the syntactic restriction given in [Knoblock *et al.*, 1991], and used as default by ALPINE if no primary effects are provided by the user. As we have demonstrated, it often creates too few number of abstraction levels.
2. **User-Defined Primary Effects.** This is the approach taken by ALPINE, and many other systems. For example, the ABTWEAK system [Yang *et al.*, 1991] depends on the user to define the set of primary effects of operators.
3. **Automatically Selecting Primary Effects.** This is the approach we are taking. We now give a more detailed description of the algorithm.

For each operator, a definition of primary effects should make a clear distinction between those effects that are important, and those that are not. A good distinction thus relies on a formalization of what is important. The notion of importance that we follow is based on the intuition of ordered hierarchies:

> Given two effects e_{pri} and e_{sec} of an operator α, e_{pri} is more important (i.e., primary) than e_{sec} if it is possible to achieve e_{sec} without violating e_{pri}, while to achieve e_{pri} one has to violate e_{sec}.

In other words, e_{sec} is easier to achieve compared to e_{pri}, because it is possible to achieve it without violating e_{pri}. As an example, one can have a pleasant walk without going to a computer shop, but to buy diskettes, one has to go to a computer shop. Thus, taking a walk is a secondary effect of going to a computer shop as compared with buying diskettes.

From Theorem 1, every choice of primary effects that satisfies Restriction 2 results in an ordered hierarchy. But different choices of primary effects give rise to different hierarchies. The finest distinction between primary and secondary effects also corresponds to an ordered hierarchy with the greatest number of abstraction levels. Therefore, when finding the primary effects of operators, we strive to maximize the total number of abstraction levels for an ordered hierarchy.

Our strategy is to augment the ALPINE algorithm by providing it with a facility of choosing primary effects. ALPINE constructs an abstraction hierarchy by building a constraint graph of literals. The literals in a problem domain are represented as nodes of a directed graph. Constraints are represented as directed edges. An edge from l_1 to l_2 indicates that $crit(l_1) \geq crit(l_2)$. Initially, the graph is a set of literals without any constraints. When the algorithm terminates, the strongly connected components of the graph correspond to abstraction levels. The abstraction levels also relate closely to the primary effects for each operator: an effect e_1 is primary if it has a criticality value no less than the other effects of the same operator. Our algorithm will thus try to leave as many strongly connected components as possible.

To avoid an exhaustive search, we use a greedy algorithm. The algorithm uses a heuristic for incrementally adding edges to the graph, attempting to impose as few constraints as possible. To do this, the algorithm processes operators which impose the fewest number of constraints first, and consider the operators which have the potential to impose a large number of constraints later. A heuristic function for determining the amount of constraints equals the total number of effects of an operator. At the same time, we also try to guarantee planning completeness, by making sure that every operator has a primary effect, and every literal is the primary effect of at least one operator.

To sum up, our algorithm first sorts the operators in ascending order of the number of their effects. While building the constraint graph and choosing primary effects, the algorithm starts by considering operators each of which achieves exactly one literal. Thus, for each operator α that has a unique effect e, we make e the primary effect of α, and add directed edges from e to all preconditions of α. Then we consider each literal achieved by a unique operator. Since each achievable literal must be a primary effect of some operator, we make every literal achieved by a unique operator a primary effect of the corresponding operator. In the next iteration, we consider the set of operators that establishes two distinct literals, and the literals that are achieved by two different operators. At the i-th step, the algorithm performs the following two operations:

- choose primary effects of each operator that establishes i different literals, and
- for every literal l that is achieved by i different operators, make l a primary effect of one of the corresponding operators.

Each choice of a primary effect will add edges to the constraint graph. Given an operator with m effects, there are m possible choices. The best choice is dictated by maximizing the total number of strongly connected components in the constraint graph. In other words, the algorithm uses a greedy strategy by making locally optimal choice at each step.

Our algorithm, *Choose_Primary_Effects*, is shown in Table 4. Its input is a set of operators in a domain, and it outputs a selection of primary effects for each operator. It gives the user the option to define primary effects of some (not necessarily all) operators, and then chooses primary effects of the remaining operators. The notation $||Graph||$ refers to the total number of strongly connected components in the graph.

We now give a more detailed description of the algorithm. Line 2 of the main algorithm adds edges defined by Restriction 2 for the user-defined primary effects. Then the algorithm chooses primary effects of the remaining operators. At the i-th step, we choose primary effects of operators that achieve i literals. This is performed by the algorithm *Choose_Prim_Effects_Of_Operator*. Let α be an operator, $e_1, e_2, \ldots, e_i$ be effects of α, and *Graph* be the constraint graph before a primary effect of α is chosen. First the algorithm tries to make e_1 a primary effect of an operator, by adding directed edges from e_1 to all other effects of α and to all preconditions of α, thus creating a new graph $Graph_1$. Then the algorithm tries to make e_2 a primary effect of α and creates the corresponding graph $Graph_2$. Similarly, it generates $Graph_3, \ldots, Graph_i$. After all graphs are generated, the algorithm counts the number of strongly connected components in each of the graphs, and chooses the graph $Graph_j$ with the largest number of components. e_j is then chosen as a primary effect of α. This operation is performed for all operators that achieve i distinct literals.

The algorithm next considers all literals achieved by i different operators, and make each literal a primary effect of one of the corresponding operators. This is performed by the procedure *Make_Literal_Be_Prim_Effect*, which is similar to *Choose_Prim_Effect_Of_Operator*. For the lack of space, we do not present it here. Interested readers can refer to [Fink, 1992].

After the *Graph* is completely built, line 10 of the main algorithm chooses the remaining primary effects of operators according to the imposed constraints: for each operator α, the criticality of every effect of α is compared with the criticality of the primary effect found by the procedure *Choose_Prim_Effect_Of_Operator*. All effects of α whose criticalities are equal to the criticality value of the primary effect, are added to the set of primary effects of α.

It can be shown that the running time of the algorithm is $O(|\mathcal{L}|^2 \cdot \sum_{\alpha \in O} |\mathit{Eff}(\alpha)|)$, where $\mathcal{L}$ is the set of all literals in the problem domain, and O is the set of all operators in the problem domain.

As an example, we consider the Extended Tower of Hanoi domain. Each of the operators *MoveL*, *MoveM*, and *MoveS* achieves one literal, and at the first step the algorithm makes this literal a primary effect. After performing this step the graph G is as shown on Figure 1a. Then the algorithm considers operators that achieve two distinct literals. These operators are *MoveLM*, *MoveMS*, and *MoveLS*. The effects of *MoveLM* are *OnLarge* and *OnMedium*. One of them must be chosen as a primary effect. If *OnLarge* is a primary effect, its criticality must be at least as great as the criticality of *OnMedium* and the criticalities of all preconditions of *MoveLM*. These restrictions already hold in G, so it is not necessary to add new constraints. If *OnMedium* is chosen as a primary effect of *MoveLM*, we must have $crit(OnMedium) \geq crit(OnLarge)$. After the constraint defined by this inequality is added to G, we receive a new graph G' shown in Figure 1b. G' contains fewer strongly connected components than G. Since the purpose is to maximize the number of strongly connected components, the algorithm finally chooses *OnLarge* to be a primary effect of *MoveLM*. Then the algorithm uses the same method to choose primary effects of *MoveMS* and *MoveLS*. One may check that the algorithm chooses *OnMedium* to be a primary effect of *MoveMS*, and *OnLarge* to be a primary effect of *MoveLS*. According to Restriction 2, the predicates *OnLarge*, *OnMedium* and *OnSmall* have criticality values 2, 1, and 0, respectively.

Table 4: Creating an Ordered Hierarchy

```
Choose_Primary_Effects
Graph := create a directed graph where
    (a)  every literal in the problem domain
           is represented as a node, and
    (b)  there are no edges between the nodes
Add_Constraints_For_User-Defined_Prim_Effects;
for i := 1 to n do
begin
    for each α that achieves i distinct literals do
        if the user have not defined prim. effects of α
            then Choose_Prim_Effect_Of_Operator(α);
    for each l achieved by i distinct operators do
        if l is not yet a prim. effect of some operator
            then Make_Literal_Be_Prim_Effect(l);
end;
Choose_Prim_Effects_According_To_Graph
Topological_Sort

Choose_Prim_Effect_Of_Operator(α)
Max_Number_Of_Comps := 0;
for each e1 ∈ Eff(α) do
begin
    Graph1 := Graph;
    for each e2 ∈ Eff(α) do
        add an edge in Graph1 from e1 to e2;
    for each p ∈ Pre(α) do
        if p can be achieved by some operator
            then add an edge in Graph1 from e1 to p;
    Combine_Strongly_Connected_Comps(Graph1);
    if ||Graph1|| > Max_Number_Of_Comps
        then
            begin
                Graph2 := Graph1;
                primary := e1;
                Max_Number_Of_Comps := ||Graph1||
            end
end;
Graph := Graph2;
Prim-Eff(α) := {primary}
```

6 ADVANTAGES AND LIMITATIONS OF USING PRIMARY EFFECTS

In this section, we discuss the advantages and limitations of an abstraction hierarchy based on Restriction 2, as compared to a hierarchy built by ALPINE. We compare two types of abstraction levels in terms of the number of hierarchies generated by each algorithm, and discuss the completeness of the resulting planning system.

First observe that the number of abstract levels generated by our algorithm is always no less than that generated by ALPINE. This is because our algorithm imposes criticality constraints only for *primary effects*, while ALPINE imposes constraints for *all* effects of an operator, unless the primary effects are provided by the user.

Second, if we need to establish some literal l, we may use only operators with a *primary* effect l, not all the operators that achieve l. This reduces the branching factor of search.

Now we discuss the *completeness* of a system based on the abstraction hierarchy produced by our algorithm. Ideally, we would like an abstraction hierarchy to have the *monotonic property.* The monotonic property holds if for every solvable problem there exists a *justified* [3] abstract-level plan that can be refined to a concrete-level plan that solves the problem. Yang and Tenenberg have shown that every abstraction hierarchy satisfy this property [Yang *et al.*, 1991]. However, this claim holds only for problem-solvers that do not use primary effects. A primary-effect restricted planner applied to an abstraction hierarchy may not satisfy the monotonic property.

[3] A plan is justified if every operator either directly or indirectly contributes to achievement of a goal. In other words, a justified plan does not contain "useless" operators.

(a) Graph G (b) Graph G'

Figure 1: Graphs In the Extended Tower Of Hanoi Example

To solve the completeness problem, we present a theorem that allows us to test whether given hierarchy (built with primary effects) has the monotonic property. Let S be a state, and α be an operator whose preconditions are satisfied in S. Let $l_1, \ldots, l_k$ be the primary effects of α, and $l_{k+1}, \ldots, l_n$ be its side effects. We say that α is *replaceable on a lower level* for an initial state S if there exists some plan Π with an initial state S such that Π achieves all *side* effects of α and leaves all other predicates unchanged. In other words, all side effects of α may be achieved on some lower level of abstraction without violating any higher-level literals and any other low level literals. In our example with the computer shop, this means that you can spend $20 and have a walk *without* bying diskettes.

Theorem 2 *A hierarchy has the monotonic property* if and only if *for every state S and for every operator α whose preconditions are satisfied in S, α is replaceable on a lower level for the initial state S.*

The reason why this theorem holds may be explained informally as follows. Suppose we have some planning problem and a concrete-level plan Π that solves this problem. Let Π' be a justified version of Π on the abstract level. Such a version always exists by the Upward Solution Property [Tenenberg, 1988]. We wish to show that Π' may be refined on the concrete level. Π may not be a concrete-level refinement of Π', since it may contain additional operators that achieve abstract-level literals. But all operators inserted into Π' are used to achieve concrete-level literals. So if some inserted operator has primary effects on the abstract level, it is inserted only for the sake of its side effects. The operator is replaceable, so we may replace it with a sequence of operators that achieve only its side effects, and leave all abstract-level literals intact. Let us replace all the newly inserted operators that have abstract-level effects with sequences of operators that achieve only their side effects. Our plan is still correct, and all operators inserted into Π' now have only concrete-level effects. So, this new plan is a concrete level refinement of Π'.

Based on this theorem, we can build an algorithm to test whether a particular selection of primary effects yield a monotonic hierarchy. It can be verified that the Tower of Hanoi domain, with the extended operators and the chosen primary effects satisfies the conditions of the theorem.

7 A ROBOT-DOMAIN EXAMPLE

In this section we demonstrate the result of applying our algorithm to a simple robot domain taken from [Yang *et al.*, 1991], which is a simplification of the domain from [Sacerdoti, 1974]. In this domain there is a robot that can walk within several rooms. Some rooms are connected by doors, which may be open or closed. In addition, there are a number of boxes, which the robot can push either within a room or from one room to another. Figure 2 shows an example of a robot domain. The domain may be described by the following predicates:

open(d)	door d is open
box-inroom(b, r)	box b is in room r
box-at(b, loc)	box b is at location loc
robot-inroom(r)	the robot is in room r
robot-at(loc)	the robot is at location loc
location-inroom(loc, r)	location loc is in room r
is-door(d)	d is a door
is-box(b)	b is a box

(Observe that the last three predicates are not achievable.) The list of operators in this domain, described on LISP, is given in Table 5.

A straightforward application of Restriction 1 to this domain fails to produce a multilevel hierarchy, while the algorithm *Choose_Primary_Effects* divides the achievable predicates of the robot domain into two abstraction levels shown in Figure 3. The primary effects of operators chosen by the algorithm are marked by "; $**$" in Table 5.

8 CONCLUSION

This paper presents an extension of Knoblock's ALPINE algorithm for automatically generating ab-

Table 5: The Operators Of the Robot World

```
---- Operators For Moving Within a Room ----
```

```
; Go to Location within room
(setq o1 (make-operator
  :name '(goto-room-loc $from $to $room)
  :preconditions '(
     (location-inroom $to $room)
     (location-inroom $from $room)
     (robot-inroom $room)
     (robot-at $from))
  :effects '(
     (not robot-at $from)  ;**
     (robot-at $to))))     ;**
```

```
; Push box between locations within a room
(setq o2 (make-operator
  :name '(push-box $box $room $box-from-loc
                  $box-to-loc robot)
  :preconditions '(
    (is-box $box)
    (location-inroom $box-to-loc $room)
    (location-inroom $box-from-loc $room)
    (box-inroom $box $room)
    (robot-inroom $room)
    (robot-at $box $box-from-loc)
  :effects '(
    (not robot-at $box-from-loc)
    (not box-at $box $box-from-loc)  ;**
    (robot-at $box-to-loc)
    (box-at $box $box-to-loc))))     ;**
```

```
---- Operators For Moving Between Rooms ----
```

```
; Push box through door between two rooms
(setq o3 (make-operator
  :name '(push-thru-dr $box $door-nm
               $from-room $to-room
               $door-loc-from
               $door-loc-to robot)
  :preconditions '(
    (is-door  $door-nm $from-room $to-room
            $door-loc-from $door-loc-to)
    (is-box $box)
    (box-inroom $box $from-room)
    (robot-inroom $from-room)
    (box-at $box $door-loc-from)
    (robot-at $door-loc-from)
    (open $door-nm))
  :effects '(
    (not robot-inroom $from-room)
    (robot-inroom $to-room)
    (not box-inroom $box $from-room)    ;**
    (box-inroom $box $to-room)          ;**
    (robot-at $door-loc-to)
    (box-at $box $door-loc-to)          ;**
    (not robot-at $door-loc-from)
    (not box-at $box $door-loc-from)))) ;**
```

```
; Go through door between two rooms
(setq o4 (make-operator
  :name '(go-thru-dr $door-nm $from-room
                $to-room $door-loc-from
                $door-loc-to)
  :preconditions '(
    (is-door  $door-nm $from-room $to-room
           $door-loc-from $door-loc-to)
    (robot-inroom $from-room)
    (robot-at $door-loc-from)
    (open $door-nm))
  :effects '(
    (robot-at $door-loc-to)        ;**
    (not robot-at $door-loc-from)  ;**
    (not robot-inroom $from-room)  ;**
    (robot-inroom $to-room))))     ;**
```

```
---- Operators For Opening and Closing Doors ----
```

```
; Open door
(setq o5 (make-operator
  :name '(open $door-nm $from-room $to-room
             $door-loc-from $door-loc-to)
  :preconditions '(
    (is-door $door-nm $from-room $to-room
           $door-loc-from $door-loc-to)
    (not open $door-nm)
    (robot-at $door-loc-from))
  :effects '(
    (open $door-nm))))  ;**
```

```
; Close door
(setq o6 (make-operator
  :name '(close $door-nm $from-room $to-room
              $door-loc-from $door-loc-to)
  :preconditions '(
    (is-door $door-nm $from-room $to-room
           $door-loc-from $door-loc-to)
    (open $door-nm)
    (robot-at $door-loc-from)
  :effects '(
    (not open $door-nm))))   ;**
```

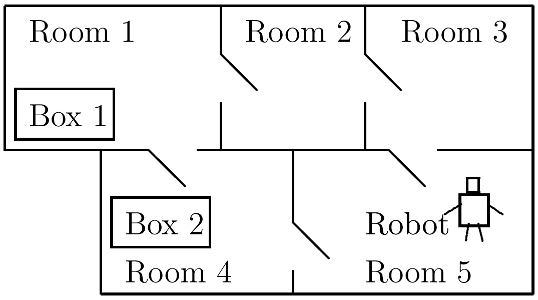

Figure 2: Example of a Robot Domain

level 1	box-inroom, box-at
level 0	robot-inroom, robot-at, open

Figure 3: Abstraction Hierarchy In the Robot Domain

straction hierarchies. Using the notion of primary effects and primary-effect restricted problem-solvers, we are able to generate ordered hierarchies where ALPINE fails. The algorithm for finding primary effects is also novel in *automatically* selecting primary effects in a given domain. We also discussed possible shortcomings and advantages of our system as compared to ALPINE.

An important extension of the described method is the algorithm that generates *problem-specific* ordered hierarchies based on primary effects. Such hierarchies are built based on individual problem instances. We have found such an algorithm, a description of which is presented in [Fink, 1992]. It allows one to generate an ordered hierarchy for a specific goal, while permitting the resulting hierarchy to be finer-grained than a problem-independent hierarchy.

The method for finding primary effects presented in the paper is purely syntactic. A possible direction of future work is to address the semantic meaning of primary effects. Another open problem is to find an algorithm that generates a hierarchy with the *maximal* possible number of levels, using the A* algorithm. Such a hierarchy will be particularly useful in a domain where the same hierarchy will be used many times.

Acknowledgements

The authors are supported in part by a scholarship and grants from the Natural Sciences and Engineering Research Council of Canada and ITRC.

References

[Fink, 1992] Eugene Fink. Justified plans and ordered hierarchies. Master's thesis, University of Waterloo, Department of Computer Science, Waterloo, Ont. Canada, Forthcoming 1992.

[Knoblock *et al.*, 1991] Craig Knoblock, Josh Tenenberg, and Qiang Yang. Characterizing abstraction hierarchies for planning. In *Proceedings of the 9th AAAI*, Anaheim, CA, 1991.

[Knoblock, 1991] Craig A. Knoblock. *Automatically Generating Abstractions for Problem Solving*. PhD thesis, School of Computer Science, Carnegie Mellon University, May 1991. Tech. Report CMU-CS-91-120.

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

[Tate, 1977] Austin Tate. Generating project networks. In *Proceedings of the 5th IJCAI*, pages 888–893, 1977.

[Tenenberg, 1988] Josh Tenenberg. *Abstraction in Planning*. PhD thesis, University of Rochester, Dept. of Computer Science, May 1988.

[Wilkins, 1984] David Wilkins. Domain-independent planning: Representation and plan generation. *Artificial Intelligence*, 22, 1984.

[Yang *et al.*, 1991] Qiang Yang, Josh Tenenberg, and Steve Woods. *Abtweak: Abstracting a nonlinear, least commitment planner*. Dept. of Computer Science, University of Waterloo, Dec. 1991. Research Report CS-91-65.