Incremental Selection in Plan Composition

John Anderson and Art Farley

CIS-TR-90-11
January, 1991

Department of Computer and Information Science
University of Oregon
Eugene, OR 97403

Incremental Selection in Plan Composition

John S. Anderson and Arthur M. Farley
Department of Computer and Information Science
University of Oregon
Eugene, OR 97403

Abstract

We view planning as a constraint-based composition problem: design a structure (i.e., a plan) by selecting and arranging elements (i.e., operators) from a pre-defined set of alternatives, such that the resultant structure satisfies certain constraints. We show how generalized operators allow for partial commitment to operator selections. Using generalized operators gives the advantage of allowing constraints to be introduced and propagated without requiring an arbitrary choice to be made. The generalizations can be incrementally specialized as propagated constraints reduce the number of alternatives.

We describe OPIE, an implemented planner that uses incremental selection. We illustrate OPIE's performance on a small blocks world example. We also show how altering the control strategy allows OPIE to replicate the behavior of a variety of previous planning systems.

1. Introduction

Previous research has shown that over-commitment can lead to unnecessary search in planning problems. For instance, linear planners commit to a particular ordering of operators whenever a new operator is added to the plan, requiring search to find the correct ordering. Non-linear planners such as NOAH [Sacerdoti 77], on the other hand, delay committing to a particular ordering until forced to do so. By eliminating the need to choose between two possible orderings, non-linear planners do less search.

Similarly, early planners frequently over-committed by selecting specific objects to fill roles in operators. Later planners such as MOLGEN [Stefik 81] delayed commitment to a particular object. Instead, the planner maintained a list of constraints that described the class of objects that could fill the role. As constraints were propagated from other design decisions, the class of possible role fillers was gradually reduced. If enough constraints could be accumulated to reduce the class of possible role-fillers to one, the choice became deterministic. Even if the number of choices was more than one, any reduction in the number of options would reduce the need for search.

One area of over-commitment that remains is selecting the operators for a plan. If more than one operator is available which achieves a particular goal or precondition, most planners must try each operator in turn. This is over-commitment in the sense that the planner commits to the first operator in the list, and may go to a lot of work developing a plan that includes that operator, only to find that the solution requires using one of the alternative operators. A planner over-commits by choosing a specific operator from a set of alternatives.

In this paper we propose a method for delaying commitment to a particular operator. The fundamental idea is the same as that used in delaying commitment to a particular object. Rather than selecting a particular operator, we use an abstract operator that generalizes all of the alternative operators that achieve the same goal. The abstract operator represents a *partial commitment*: a commitment to a class of operators, but not to a particular operator within that class.

Our method is closely tied to our representation of plans. Therefore, a relatively large portion of the paper is devoted to describing key aspects of that representation. We then describe the basic search algorithm used by our system, named OPIE. This is followed by an example, demonstrating the system as it solves the Sussman Anomaly from the blocks world.

OPIE can be considered a planner 'shell' in the sense that it can replicate the behavior of most well-known planners by adjusting its control strategy. We describe how OPIE can be tailored to mimic other planners. Finally we discuss the strengths and limitations of our approach.

2. Representation

We view planning as a composition process. Our approach is based on the notion of search in a plan space [Charniak et al. 87], [Chapman 87], [Georgeff 88]. A plan is gradually composed by selecting components from a pre-defined set and arranging them so as to satisfy a set of constraints. In our view a plan describes three kinds of entities: *objects*, *persistences*, and *actions*.

An **object** is anything which can be acted upon.

A **persistence** is an n-ary relation between objects that holds over some continuous period of time [McDermott 82, Dean and McDermott 87]. We treat persistences as individuals, making it possible to distinguish between different occurrences of the same relation. For example, if block A is on block B, then is removed, and later is replaced, two separate persistences of ON(A,B) have occurred. This is slightly different from approaches that treat the *fact* ON(A,B) as an entity that can hold over discontinuous intervals [McDermott 82, Shoham 87, Chapman 87].

Finally, an **action** is any event in which a persistence is produced or consumed. In this paper, we only discuss actions under the control of the planner, called *operators*.

We also distinguish between *types* and *instances*. An **instance** is an individual object, persistence, or operator. A **type** is a class of individuals that share a set of common characteristics. The defining characteristics of a type are the possible relations between instances of that type and of other types.

In the blocks world domain which we will use to illustrate our system, the object types are BLOCK and TABLE. Figure 1 presents the persistence and operator types for the domain. Objects are defined by the persistences and operators in which they may be in-

Clear
Predicate: Clear
Objects: Block
Producers: BB, BT

On1
Predicate: On
Objects: Block, Table
Producers: BT

On2
Predicate: On
Objects: Block, Block
Producers: TB, BB

Figure 1a. Persistence types for the blocks world domain.

Move-Block-from-Table-to-Block (TB)
Objects: Block,Table,Block
Consumes: On1(1,2), Clear(3)
Uses: Clear(1)
Produces: On2(1,3)

Move-Block-from-Block-to-Block (BB)
Objects: Block,Block,Block
Consumes: On2(1,2), Clear(3)
Uses: Clear(1)
Produces: Clear(2), On2(1,3)

Move-Block-from-Block-to-Table (BT)
Objects: Block,Block,Table
Consumes: On2(1,2)
Uses: Clear(1)
Produces: Clear(2), On1(1,3)

Figure 1b. Operator types for the blocks world domain.

volved. Persistences are defined in terms of the objects they relate and the operators that can produce and consume them. OPerators are defined in terms of the objects they involve and the persistences they produce, consume, and use. An operator *produces* a persistence if the corresponding relation becomes true during the action. An operator *consumes* a persistence if the cooresponding relation becomes false during the action. An operator *uses* a persistence if the corresponding relation must be true for the action to occur, is true throughout the action, and continues to be true after the action.

It may seem redundant to have two ON persistence types: one for blocks on blocks and one for blocks on tables. However, this allows us to specify exactly what can and cannot be stacked: a table is not allowed to be on a block or on another table.

Temporal information about a persistence may be given by a pair of time points indicating the time the persistence begins and ends [Dean 88]. Alternatively, the beginning time point and the duration of the persistence may be used [McDermott 82]. In this paper, we are not dealing with scheduling problems, and therefore are not concerned with the exact time points at which persistences begin and end. However, we do need to keep track of which operators produce and consume each persistence.

THe objects involved in a persistence or operator type appear in an ordered list. We use the position of the argument in the list as the local name of that object. For example, operator TB would be interpreted as follows. For a block (1) to be moved from a table (2) to a block (3), block (1) must be on table (2) and clear, and block (3) must be clear. The effect of the operator is that block (1) is no longer on table (2), block (3) is no longer clear, and block (1) is on block (3).

We refer to a symbolic representation of a type or instance as a *designator*. Our designators are like frames, with slots indicating relations between entities. For any single problem, the set of type designators is fixed. Instance designators, on the other hand, are created and manipulated during the problem solving process. Type designators serve as templates for instance designators and also serve to constrain the potential slot fillers.

For example, a persistence of type ON1 would have two object slots: one requiring an object of type BLOCK and the other requiring an object of type TABLE. It would also have a producer slot requiring an operator of type BT. Note that persistence instances differ from persistence types in that a persistence instance can have only one producer and only one consumer, but a persistence type can have any number of *possible* producing and consuming operator types.

Traditional STRIPS-style planners rely on adding and deleting propositions from a state description to indicate changing beliefs about the state of the world. In the plan composition approach, however, nodes do not contain a description of the world, but some version of the plan. While the 'produces' and 'consumes' lists contain the same kind of information as STRIPS add and delete lists, the interpretation is different. 'Consumes' and 'produces' are relations between entities described by the plan (i.e., between an op-

erator and a persistence). Adding an operator to the plan which consumes a persistence does not remove the persistence from designator from the plan, it simply fills in the 'consumer' slot of the designator. This is a key aspect of composition problems: we are monotonically adding information to the solution description. Our ability to utilize constraints to prune the search space relies on this monotonic nature of the composition process.

3. Plan Composition

Persistence designators with empty producer slots play a special role in planning. These unproduced perisitences represent goals of the planner: relations which need to be achieved, but are not yet attributed to a particular producer. The primary task of a planner is to find a producer for each unproduced persistence.

The traditional definition of planning is to find a sequence of operators which achieves a goal from a given initial state. The input is a set of candidate operator types, a description of the goal, and a description of the initial state. OPIE takes the same input as other planning programs. OPIE treats the goals as a set of unproduced persistences which are consumed by a *final consumer* and the initial state as a set of unconsumed persistences produced by an *initial producer*. The initial producer and final consumer are treated as operators during the planning process: the first and last operators of the plan. OPIE's task is to produce a plan, consisting of operators, persistences, and objects, which contains no unproduced persistences. Each persistence in the plan must be *attributed* to an operator in the plan (possibly to the initial producer).

OPIE searches in the space of plans. We use the term *node* to refer to a snapshot of the plan under construction. We refer to plan modification operations as *node refinement*. The basic plan composition algorithm is:

```
0) Create a node containing the initial producer and final
   consumer and place it in the search queue.
1) Select a node from the search queue.
   1.a) If the search queue is empty, fail.
   1.b) If the node satisfies the success criteria,
        report success and return the node.
2) Refine the node:
   2.1) Select a plan component to refine.
   2.2) Generate a child node for each possible refinement.
   2.3) Complete each new node by propagating constraints.
3) Evaluate each new node:
   3.1) If any constraint is violated, reject the node.
   3.2) Else add the node to the search queue.
4) Go to step 1.
```

Every plan cycle starts with the selection of a node to be refined. We use a simple best-first search strategy, where the value of a node is the number of operators in the plan. If two nodes have the same number of operators, the one with the fewest unproduced persistences is selected. If two nodes have the same number of operators and unproduced persistences, the one generated most recently is selected (LIFO). OPIE can employ additional heuristics to focus the search; however, for purposes of this paper, we ignore them.

Once a node has been chosen, some plan element in that node must be selected to be refined. One refinement method is to *attribute* an unproduced persistence, that is, find a producer for it. There are two types of attribution: *simple attribution*, in which the producer is an operator already in the plan, and *attribution with operator insertion*, which requires that a new operator be added to the plan. We will refer to the latter as 'operator insertion' or simply 'insertion'.

Potential producers already in the plan are found by comparing the unproduced persistence with all of the persistences in the plan that have a producer but no consumer. If a match is found, the two persistences are merged - a single designator is formed from the unification of the two matching persistence designators. Thus, the previously unproduced persistence acquires a producer in the merger. If there is more than one possible producer in the plan, a separate node will be generated for each simple attribution.

Potential producers of a persistence are also found by finding all of the types of operators that produce that type of persistence. For each relevant operator type, a separate node is spawned by inserting a new operator of that type into the plan. This is really means-ends analysis: if a persistence has no producer, insert an operator which produces it. For completeness, OPIE creates nodes using operator insertion even when a producer can be found in the plan.[1]

Figure 2 presents a simple illustration of the composition process. The three operator types are shown in figure 2a. A key is given in Figure 2b. The problem to be solved is to stack block A on block B, given that both blocks are on the table and clear. The root node, with only the initial producer (IP) and final consumer (FC), is shown at the top of Figure 2c. The only refinement of that node is to attribute the unproduced ON2 (A, B) persistence. No produced persistence designators in the plan unify with the unproduced persistence designator, but two operator types can produce ON1: Move-Block-from-Block-to-Block (BB) and Move-Block-from-Table-to-Block (TB). In addition, BB can be instantiated in several ways.

Here is the situation we would like to avoid: two (or more) operators are equally likely candidates for selection. Each possible producer, whether an operator already in the plan or a new operator being introduced, results in a separate node being generated in the search space. This is one source of combinatorial explosion: each possible attribution represents an alternative branch in the search space. In a more complex example, there may be a great deal of additional search required before one option can be shown to be the correct choice. The planner is forced to guess, and backtrack if the guess turns out to be wrong. We next describe how partial commitment helps avoid the need to make guesses.

1. In section 7 we show that the part of OPIE described so far is equivalent to TWEAK.

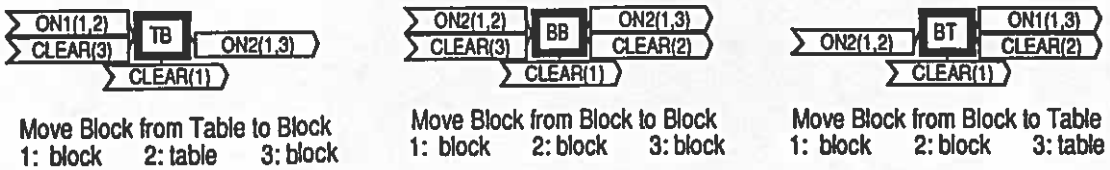


Figure 2a. Blocks-world operator types.

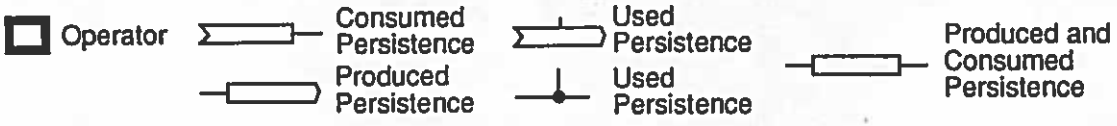


Figure 2b. Key.

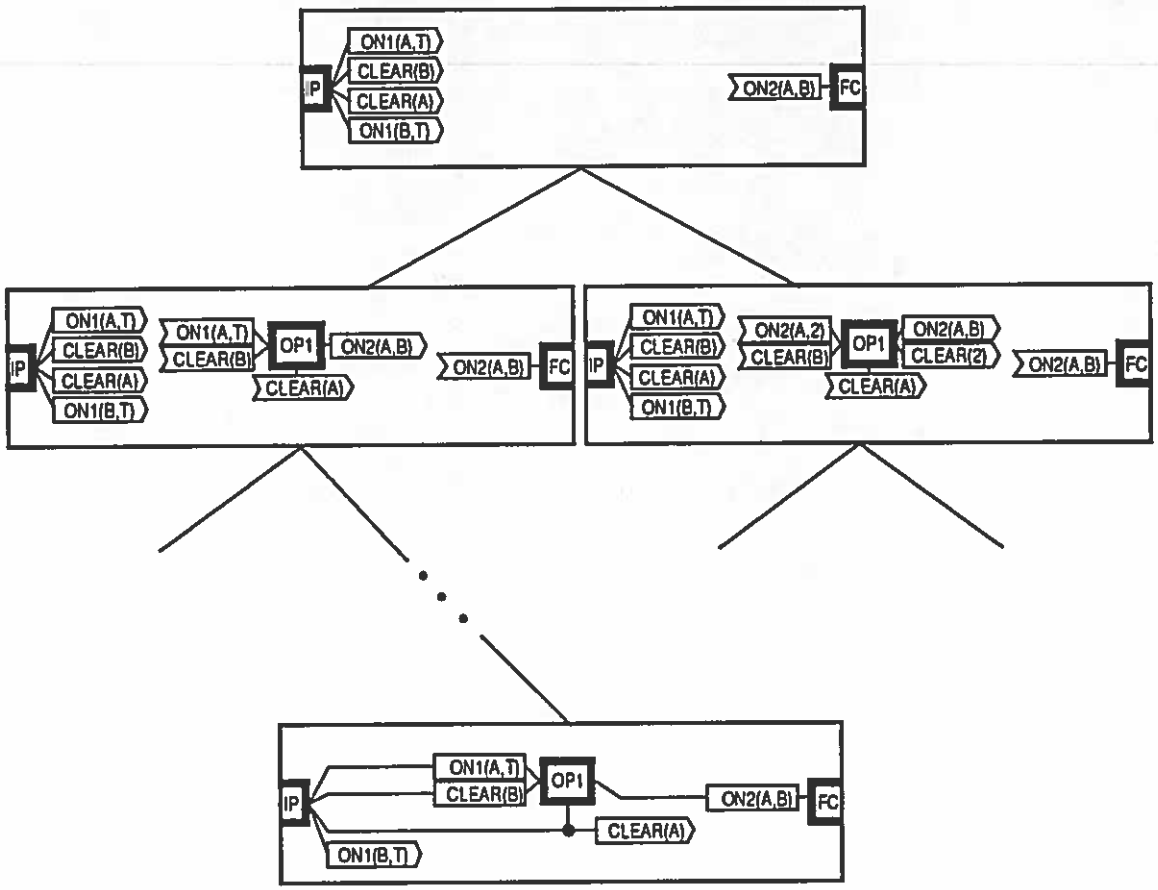


Figure 2c. Partial view of search space for achieving ON(A,B).

4. Partial Commitment via Generalization

The exponential nature of search results from indeterminacy: if we always knew which choice to make, there would be no backtracking and search would be linear in the depth of the solution. Frequently, making a commitment at one choice point introduces constraints that prune one or more options at another choice point. If the constraints are introduced early enough, i.e., before one of the pruned choices is selected, some unneccessary search is avoided. Thus, for some problems, the difference between linear and exponential search is the order in which commitments are made.

The problem is that the only way to introduce new constraints is to make a decision. This is catch-22: we want to delay making a decision until constraints can be propagated, but constraints can only be propagated if a decision is made.

In some cases, there is a solution to this dilemma. It may be possible to make a *partial commitment* to a set of alternatives. Partial commitment allows some of the consequences of a decision to be explored. Partial commitment to a class of operators allows the planner to go on to other planning decisions. If we are lucky, the constraints introduced by partial commitment will be enough to determine the right choice for another decision. The outcome of that decision, in turn, may rule out some of the alternatives for the first decision. For example, rather than selecting a particular main course right away, one could decide to eat seafood, order white wine, and then think about which fish dish to order. Then, the price of the wine might eliminate some of the higher-priced seafood items on the menu from consideration.

To be the most useful, partial commitment requires two things. First, there must be a way to represent the set of operators that are candidates for inclusion in the plan. Second, there must be a way of indicating the shared requirements of those operators. This is what makes partial commitment better than no commitment: any requirement common to all of the candidate operators becomes a constraint on the plan, allowing planning to go on 'around' the nondeterminant choice point. One way to accomplish both of these conditions is to use abstract operators to represent the common aspects of a class of operators.

The first step in our approach to partial commitment is creating abstract operator types from a given set of primitive operators. This is accomplished by organizing operators into classes, where a class consists of operators that share one or more persistences on their respective produces, consumes, and uses lists. An abstract operator is created which embodies the common aspects of members of the class. This operator becomes the parent of the members of the class. The result is a taxonomic hierarchy, with primitive operator types as the leaves and abstract operator types as the internal nodes.

Figure 3 presents the taxonomy of operators formed from our three primitive blocks world operators. The next two sections describe how OPIE automatically forms a taxonomy of operator types from a set of primitive types.

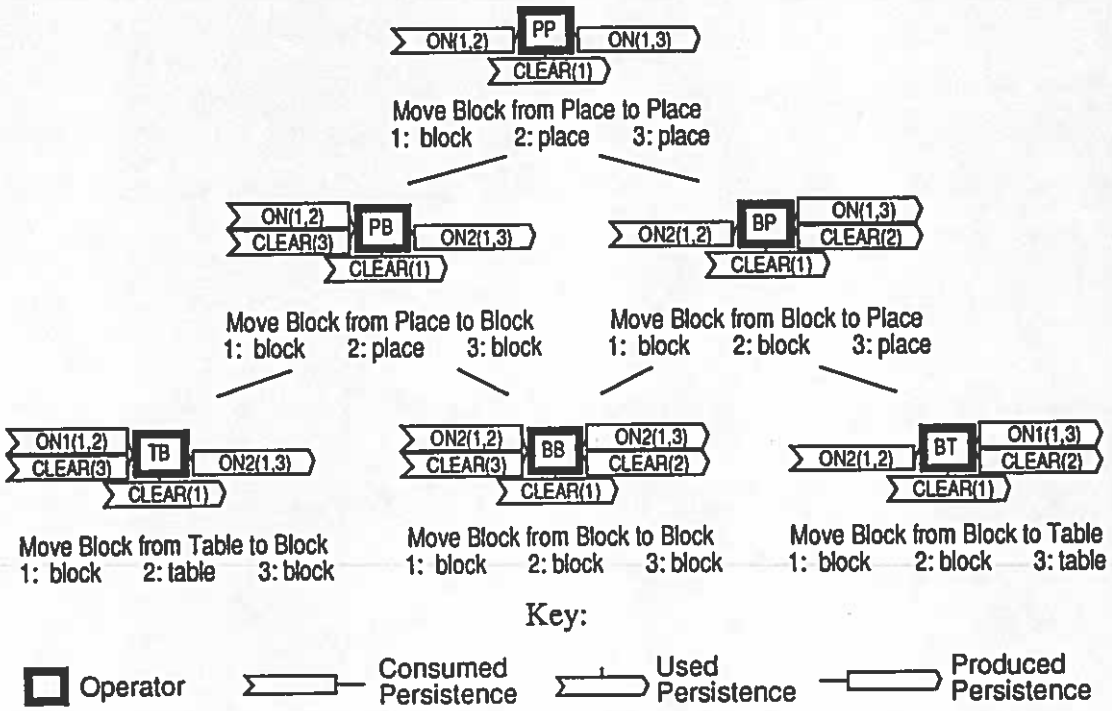


Figure 3. Taxonomy of blocks-world operators.

4.1 Pairwise generalization

Pairwise generalization extracts the commonalities of two operators to form a generalization of the pair. The possible commonalities are matching persistences on the produce, consume and use lists of the two operators. Persistences match if they have the same predicate name.

For example, consider the operators TB and BB in Figure 3 . Generalizing these two operators gives us PB which contains their common persistences. First, look at the used persistences. Both TB and BB use a CLEAR persistence, so their generalization must also produce a CLEAR persistence. Next, look at the produced persistences. ON2 is produced by both TB and BB, but BB also produces a CLEAR persistence. The generalization produces an ON2 persistence, but not a CLEAR persistence.

Finally, look at the consumed persistences. CLEAR is consumed by both operators, and is therefore consumed by their generalization. Both operators also consume a persistence whose predicate name is ON, but TB consumes ON1 and BB consumes ON2. A new persistence type is created as the generalization of ON1 and ON2. For convenience, in this paper we have assigned the name ON to the new persistence. The general operator PB consumes a persistence of type ON.

Whenever two corresponding objects in matching persistences are not of the same type, an abstract object type is also formed. For example, object 2 in TB is a TABLE and object 2 in BB is a BLOCK. These two objects correspond by virtue of being the second argument of the matching consumed ON persistences. A new object type is created as the generalization of TABLE and BLOCK. This object type is used in the new general operator PB. For this paper, we have assigned the name PLACE to the abstract object type representing things-that-a-block-can-be-on.

4.2 Forming the taxonomy

Pairwise generalization forms the core process in generating a taxonomic hierarchy. Each primitive operator is added to the hierarchy one at a time. Since we are interested in grouping operators that produce the same persistences, each new operator is compared with every previous operator that shares at least one persistence on its produce list.

The algorithm for adding new operators to the hierarchy is shown below.

```
For each operator NEW in the input set:
    Let set S be those leaf operators of the hierarchy which
    share at least one add relation with NEW.
      For each operator OP in S,
        Let G-OP := PWG(NEW,OP) and
          INSERT-ANCESTOR(G-OP,OP).
```

PWG is the pairwise generalization process described above. INSERT-ANCESTOR is used to place the new generalized operator into the hierarchy. We know G-OP is a generalization of OP, so we begin by looking at the parents of OP.

```
Define INSERT-ANCESTOR(G-OP,OP).
    For each parent P-OP of OP,
     Case:
      1. G-OP = P-OP,
         Then MERGE(G-OP,P-OP).
      2. G-OP is a specialization of P-OP,
         Then LINK-PARENT-CHILD (P-OP,G-OP).
      3. G-OP is a generalization of P-OP,
         Then LINK-PARENT-CHILD (G-OP,P-OP) and
              INSERT-ANCESTOR (G-OP,P-OP).
      4. G-OP and P-OP share persistences,
         Then Let G-P-OP := PWG(G-OP,P-OP) and
                INSERT-ANCESTOR (G-P-OP,P-OP).
      5. {G-OP and P-OP have no shared persistences}.
```

MERGE makes all children of GEN into children of AB and discards GEN. LINK-PARENT-CHILD makes its first argument a parent of its second argument.

The major points of the algorithm are illustrated in the following example of how the operator hierarchy shown in Figure 3 was constructed.

Suppose TB and BB are the first operators to be added. We have already seen that TB and BB produce PB as a generalization. Next, BT is added. Suppose BT is first compared with TB. Their generalization is PP, as shown in Figure 4.

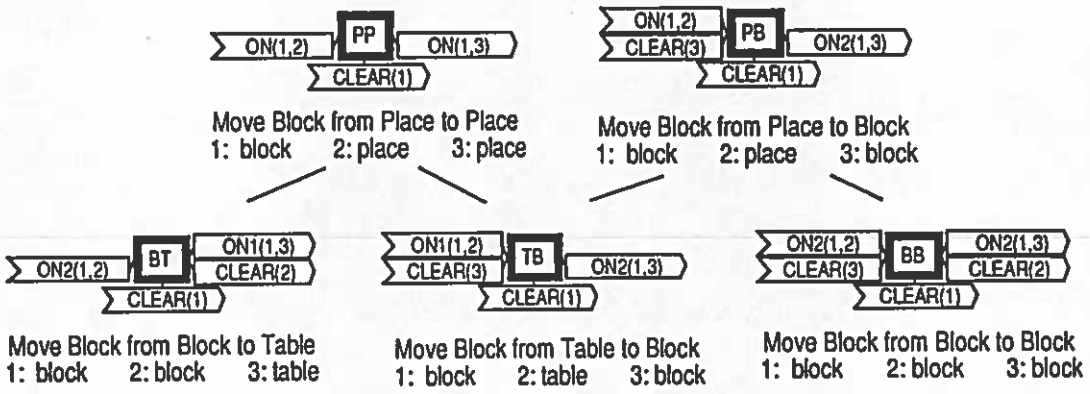


Figure 4. Building the taxonomy of blocks-world operators.

Now PP is inserted into the parents of TB. The only parent is PB. In this case, PP is found to be a generalization of PB. PP is made a parent of PB. In addition, the direct parent-child links from PP to BT and TB are dropped, since they are made redundant by the indirect links through PB. This step guarantees that the same hierarchy is always formed for a given set of primitive operators, regardless of the order in which they are added to the hierarchy.

Next, BT is compared with BB. Their generalization is BP. BP is inserted into the parents of BB. The only parent is PB. BP and PB are compared and have persistences in common. Let their generalization, which is equivalent to PP, be called PP'. PP' is inserted into the parents of PB. The only parent is PP. PP' is equivalent to PP, so the children of PP' are made children of PP and PP' is discarded. The result is the operator hierarchy shown in Figure 3.

4.3 Summary: Generalization

OPIE automatically creates abstract operator types while organizing a given set of primitive operators into a taxonomic hierarchy. During this process, the system also creates categories of abstract object types which allow abstract operators to apply to broad classes of functionally similar objects. The input to OPIE is no different from the input to other planners: an initial state description, a set of goals, and a set of primitive operator types.

5. Using abstract operators in planning

The planning task must be revised to reflect the use of abstract operators. The success conditions for a finished plan have an additional restriction. In addition to requiring that every persistence in the final plan have a producer, we also require that every operator be primitive.

We now describe how abstract operators and objects are used for partial commitment in plan composition. Recall that when a persistence is attributed, one option is to introduce a new operator into the plan. By tracing from the persistence instance to its type and then to the producers of that type, we find all of the possible operators that might be added to the plan.

Using partial commitment, however, we do not need to create a separate node for each alternative operator type. For any persistence type, there is one 'most general producer' in the operator taxonomy. It is the operator highest in the hierarchy that produces that type of persistence. This operator has as descendants every primitive producer of the persistence type. For example, all of the producers of ON2 are descendants of PB in Figure 3. By introducing an instance of an abstract operator type into the plan, we have partially committed to one of its descendants. Thus, we create only a single new node, instead of a separate new node for every primitive producer.

Each abstract operator must be specialized to a primitive operator before the plan is complete. In the worst case, the planner will have to guess at which specialization is correct, resulting in no reduction in search. However, in many cases it will be possible to work on other parts of the plan once a partial commitment has been made. By making commitments (or partial commitments) at other choice points in the plan and propagating constraints, it is possible that the set of specializations will be reduced. If so, part of the search space will have been pruned without search.

We require an additional node refinement method, that of *specializing* an abstract operator. Specialization consists of changing an operator instance from a general type to a more specific type. For example, an instance of type PB could be specialized to either type TB or type BB. A separate node is created for each specialization.

Simple attribution is slightly different once abstraction is introduced. Not only should objects match objects of the same type; they should also match objects of a more general or less general type. For example, an object of type PLACE should match blocks or tables. The complete unification algorithm is as follows:

```
1. Match an unproduced persistence designator with each unconsumed
   persistence designator which has the same predicate name.
2. Corresponding object designators from the two object lists
   match if:
      a. they are the same designator, or
      b. at least one is not bound to a particular object, and
          - they are of the same type, or
          - one's type is a taxonomic ancestor of the other's type.
3. If the entire match succeeds, replace each pair of matching
   persistence and object designators with a single new
   designator.  The new designator's type is the more specific of
   the two types.  Wherever either of the two designators appeared
   in the plan description, replace it with the new designator.
```

Attributing a persistence places temporal constraints on the operators involved. The producer of a persistence must precede the consumer. An operator that uses a persistence must follow the producer of the persistence and precede its consumer.

One decision which we have not yet mentioned is selecting which plan element to refine (i.e., an unproduced persistence to attribute or an abstract operator to specialize). Note that this is a separate decision from which node to refine, and has a separate strategy. One strategy for selecting which component to refine is the *least branches / front-to-back* (LBFB) strategy. Potential refinements are evaluated as follows:

1) Apply a refinement that will produce the fewest new nodes.
2) If there is a tie, attribute a persistence before specializing an operator.
3) If there is still a tie, prefer persistences that appear earlier in the plan.
4) If there is still a tie, make an arbitrary selection from those tied.

The least branches / front-to-back strategy is based on two ideas for reducing search. First, the least branches heuristic is intended to reduce the branching factor of the search, which directly affects the size of the search space explored [Korf 87]. Second, the front-to-back heuristic is intended to take advantage of existing constraints in the problem definition. There are two sources of constraints: the goals and the initial conditions. OPIE uses goals as constraints during attribution. The front-to-back heuristic propagates constraints from the initial conditions to guide the specialization process.

The actual strategy used by OPIE can be set by the user. By changing the strategy we can replicate the behavior of many well-known planners, as described in Section 7, below.

6. Example: Sussman Anomaly

We trace OPIE's solution of a classic planning problem known as the Sussman Anomaly [Sussman 75]. The initial and goal states are presented in Figure 4. OPIE begins by forming a plan node from the goals and initial conditions of the problem statement. This node is the root node of the problem space. The root node contains an initial producer, IP, and a final consumer, FP, which represent the problem constraints. The root node for the Sussman anomaly problem is shown below.

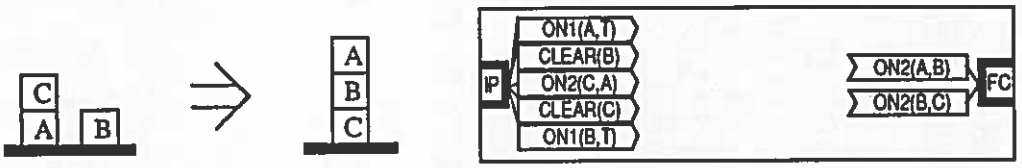
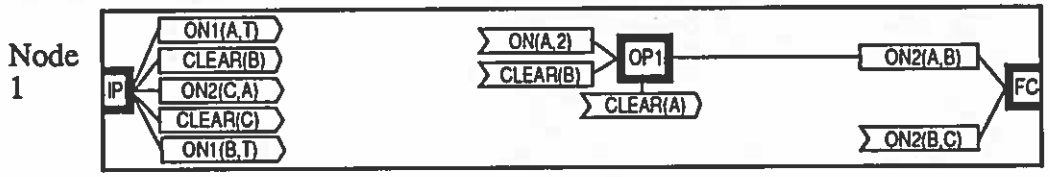


Figure 4. The Sussman Anomaly problem.

Since initially the root node is the only node in the queue, it is selected for refinement. It has two unproduced persistences, ON2 (A, B) and ON2 (B, C), and no abstract operators. There are two ways to refine the plan: attribute ON2 (A, B) or attribute ON2 (B, C). In either case, the only option is to add a new operator to the plan. Since each refinement is deterministic, it does not matter which comes first.

OPIE arbitrarily selects the persistence ON2 (A, B) to be attributed first. Since there is no matching produced persistence in the plan, the only way to attribute the persistence is to add a new operator to the plan. The most general operator type for achieving an ON2 relation is PB. OP1, an instance of type PB, is inserted into the plan. It is made the producer of ON2 (A, B) as shown in Node 1.

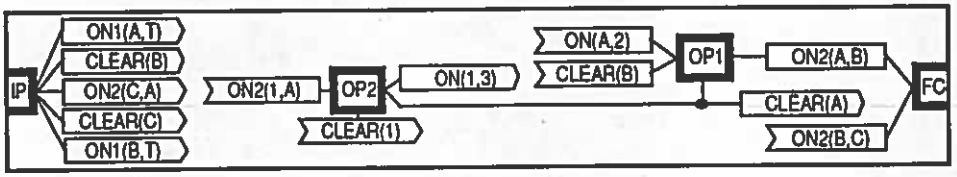


Since PB generalizes all of the operators that produce an ON2 persistence, there are no other nodes created. Therefore, the only node on the search queue is Node 1. There are four unproduced persistences in Node 1: ON (A, 2), CLEAR (B), CLEAR (A), and ON2 (B, C). In addition, there is now one abstract operator (OP1) with two possible specializations: to type TB or to type BB. There is a single matching produced persistence for ON (A, 2): ON (A, T), produced by IP. IP also produces a CLEAR(B) persistence. The other two unproduced persistences do not match any produced persistences.

The point of partial commitment is to avoid guessing. Of the five components needing refinement, three require guessing and two do not. Specializing OP1 requires guessing which specialization to choose. Attributing ON (A, 2) or CLEAR (B) require guessing whether to attribute to IP or to introduce a new operator. Attributing CLEAR (A) or ON (B, C), on the other hand, would be deterministic. In both cases, the only option is to add a new operator to the plan. OPIE selects CLEAR (A) to be attributed because it comes earlier in the plan (since OP1 must precede FC).

An instance of BP is created, since that is the most abstract operator type which produces a CLEAR persistence. The new operator is labelled OP2 in Node 2. OP2 is constrained to precede OP1 because the producer of a persistence must precede any users. Node 2 is now the only node on the search queue.

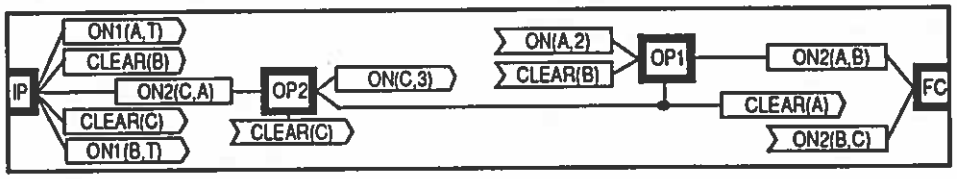
Node 2



There are now five possible attributions and two possible specializations. None of the choices are deterministic. Each of the unproduced persistences could be attributed to either an operator already in the plan or to a new operator, and each abstract operator has two possible specializations. Thus, we see that partial commitment does not guarantee a deterministic, linear solution; rather, it is a tool that must be used with other tools to reduce search.

CLEAR (1), used by OP2, can be attributed to IP in two ways: it matches either CLEAR (B) or CLEAR (C). ON2 (1, A), ON (A, 2) and CLEAR(B) can each be attributed to IP in one way. ON2 (B, C) can be attributed to OP2: it matches ON (1, 3). Using the least branches / front-to-back (LBFB) strategy, OPIE selects ON2 (1, A) to be attributed. Two nodes are created as a result of this refinement. In Node 3, ON2 (1, A) is unified with ON2 (C,A). In the other node, an operator of type PB is inserted into the plan.

Node 3

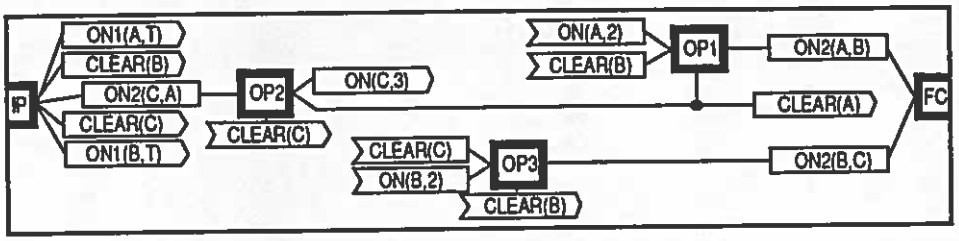


There are now two nodes on the search queue. Node 3 contains two operators, the other node contains three operators. Node 3 is selected as the best option.

Note that in attributing ON2 (1, A), object designator 1 was unified with designator C. This is propagated to the other persistences of OP2. Thus, it is no longer possible to attribute

ON2 (B, C) to OP2. This means that attributing ON2 (B, C) is now deterministic: a new operator must be inserted. Node 4 shows the result of this refinement.

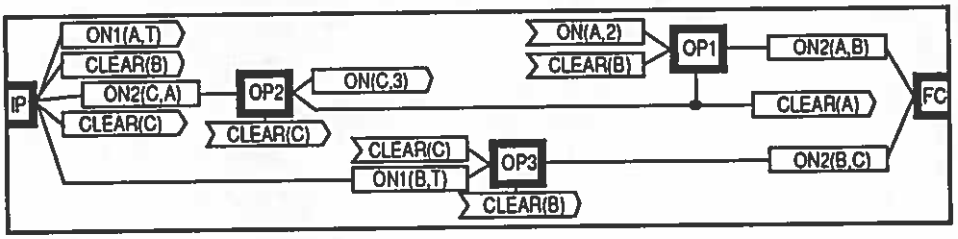
Node 4



Eventually, all of the remaining unproduced persistences will be attributed to IP. In each case, a non-solution node will also be generated with an unnecessary operator. None of these non-solution branches will be explored, because they all have more operators than the correct solution.

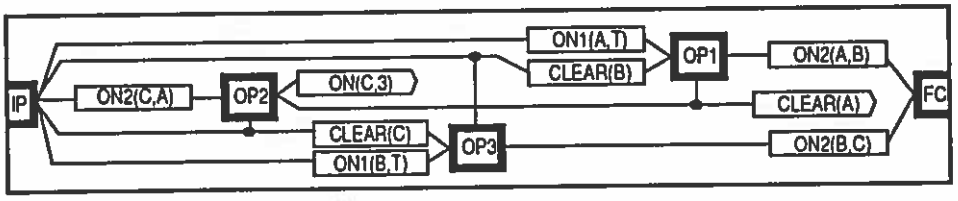
When ON (B, 2) is attributed to IP it is unified with ON1 (B, T), with 2 unified with T. Now partial commitment pays off. When 2 is unified with T it becomes type TABLE instead of PLACE. When this happens, only one of the specializations of OP3 is a valid option (refer to Figure 3). BB is eliminated because it requires that object 2 be a BLOCK. Therefore, OP3 is specialized to be of type TB. The result is shown in Node 5.

Node 5



Next, CLEAR (C) used by OP2 and CLEAR (C) consumed by OP3 are attributed to IP. This forces OP2 to precede OP3, since a user of a persistence must precede the consumer. Similarly, CLEAR (B) is used by OP3 and consumed by OP1, so OP3 precedes OP1. The plan is now completely ordered. Finally, ON (A, 2) is attributed to IP and OP1 is specialized to be of type TB, just as OP3 was specialized above. The result of these six refinements is shown in Node 11.

Node 11



The only remaining refinement is to specialize OP2. At this point, the plan says to put C somewhere and then put B on it and then A on B. For the plan to be executable, we must decide where to put C. OP2 can be specialized to either BB or BT (refer to Figure 3). If it is specialized to BB, a new unproduced CLEAR persistence will be introduced. On the other hand, if OP2 is specialized to type BT, no new persistences are introduced. Since our best-first heuristic prefers specializations which do not introduce new requirements, this specialization is selected. Therefore, the option of moving C to the table will finally be selected.

In this example, we see both the power and the limitations of partial commitment in plan composition. Some decisions can be delayed long enough for all but one alternative to be pruned, e.g., attributing ON2 (B, C) and specializing OP1 and OP3. This directly reduces the size of the space to be searched. However, it is not always possible to find a deterministic choice point using partial commitment. In addition, even if we delay a choice until the end, there may still be more than one option to explore. Thus, heuristics and/or domain knowledge are still generally required to avoid search completely. However, partial commitment is domain independent; it can be used when domain knowledge is not available. As we have seen, partial commitment can be combined with relatively general heuristics to reduce to linear complexity some problems that would be exponential with either one alone.

7. Replicating the behavior of other planners

We have been able to replicate the behavior of well-known AI planning systems by altering the refinement methods and control strategy employed by OPIE. By implementing various planners in a single planner 'shell', we are able to examine the similarities and differences among these systems. In the following discussion, we focus on the commitments made by each system during the planning process.

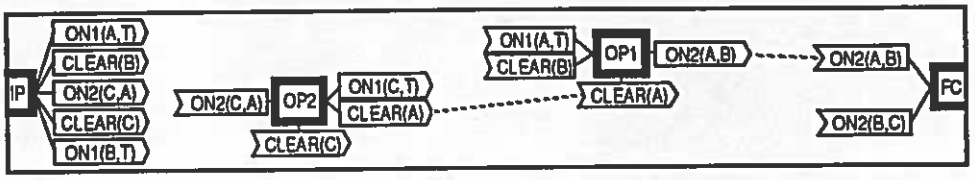
7.1 Non-specializing planners

In this section we discuss STRIPS, HACKER, ABSTRIPS, INTERPLAN, Waldinger's planner, NOAH and TWEAK. None of these planners use specialization as we have defined it, that is, as a method of selecting among alternative operators. They can be modelled using variations of attribution.

STRIPS [Fikes *et al.* 72] uses two kinds of refinement: *operator insertion* and *operator application*. In STRIPS, operator insertion adds a producer of an unproduced persistence to the plan, but does not attribute the unproduced persistence. For example, in Node 2

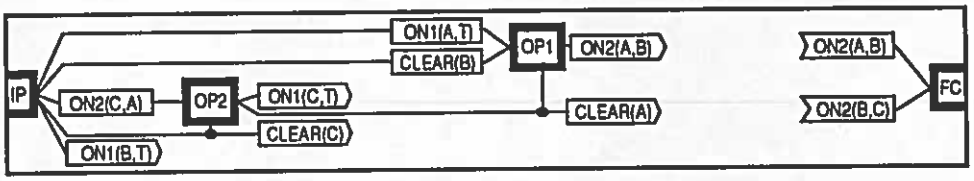
two operators have been introduced but no persistences have been attributed. This means that a goal can be achieved early in the plan but undone before it is needed.

Node 2



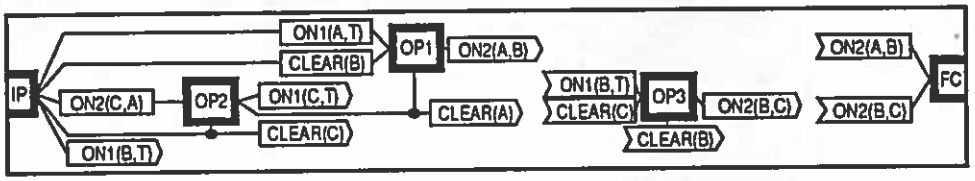
When all of the preconditions (i.e., used and consumed persistences) of an operator can be attributed by simple attribution, they are all attributed at once by operator application. STRIPS always considers the earliest unproduced persistences in the plan. OP2 is applied in the initial state; OP1 is applied in the resultant state after applying OP2. The result is shown as Node 4.

Node 4

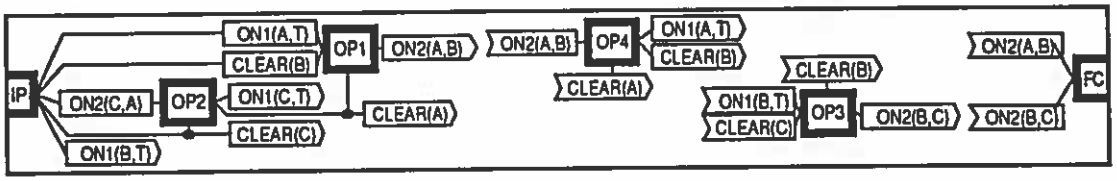


If an unproduced persistence can be attributed by simple attribution, it is not considered a goal in STRIPS. Since a produced instance of ON2 (A, B) exists, ON2 (A, B) is not a current goal. STRIPS addresses the other unproduced persistence, ON2 (B, C). OP3 is added to the plan to produce this persistence. STRIPS always inserts a new operator immediately preceding the persistence being addressed, as seen in Node 5.

Node 5



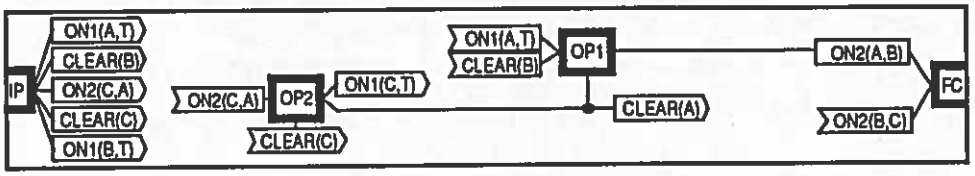
The CLEAR (B) persistence used by OP3 has no available produced persistence in the plan. The CLEAR (B) persistence produced by IP is consumed by OP1. Since STRIPS cannot put OP3 in front of OP1, a new operator, OP4, is added to produce another instance of CLEAR(B).



OP4 consumes the produced ON2 (A, B) persistence. Another operator will have to be added to produce another instance of ON2 (A, B). Thus, STRIPS fails to find an optimal solution to the Sussman Anomaly.

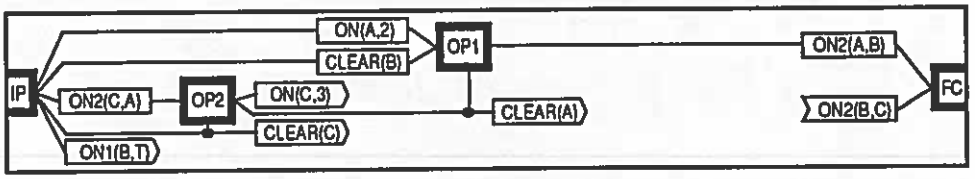
HACKER [Sussman 75] is like STRIPS, except that *operator insertion* is always accompanied by *attribution*, as in OPIE. Sussman's use of *protected goals* is subsumed by our notion of persistences. Node 2 shows the first two operators inserted in the plan. Unlike STRIPS, the goals are attributed to their producers.

Node 2



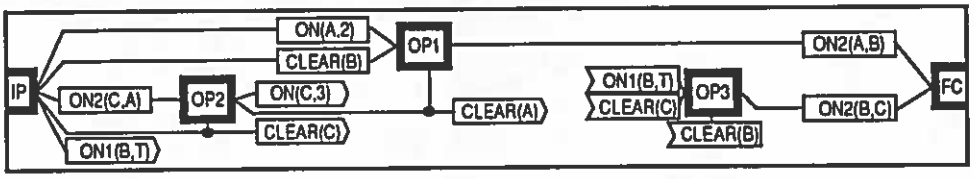
Now the operators are applied, as in STRIPS. The result is shown in Node 4..

Node 4

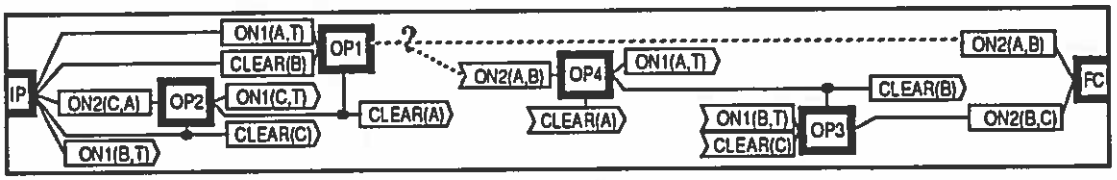


The only remaining goal is ON2 (B, C). A new operator is introduced. However, HACKER makes the same mistake as STRIPS, placing the operator *after* any operators that have been applied. The result is shown in Node 5.

Node 5

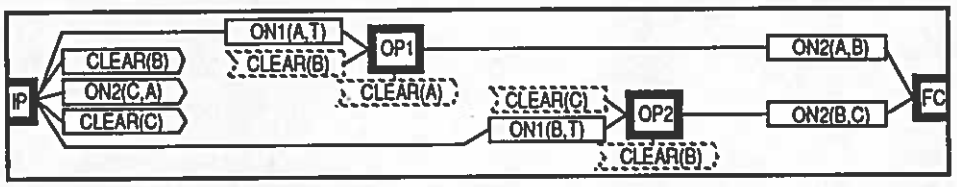


As with STRIPS, another operator must be introduced to produce a new instance of CLEAR (B). This operator is placed within the protection interval of ON2 (A, B). Since it consumes ON (A, B), a *protection violation* is detected. If a new consumer is inserted between the producer and consumer of a persistence, an anomalous situation occurs. Rather than continuing, as STRIPS would do, HACKER tries addressing the goals in a different order. Unfortunately, the alternative, solving ON (B, C) before ON (A,B), also results in a protection violation. HACKER is unable to solve the Sussman Anomaly without allowing a protection violation.



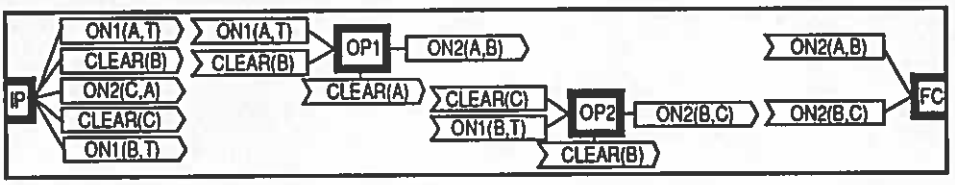
ABSTRIPS [Sacerdoti 74] uses a domain-specific ordering of precondition *criticality* to determine the order in which persistences should be attributed. For our example, suppose ON persistences are considered to be of higher criticality than CLEAR persistences. All unproduced ON persistences are attributed before any unproduced CLEAR persistences. Node 2 shows the plan for producing all of the ON persistences.

Node 2



When ABSTRIPS begins to attribute CLEAR persistences, OP1 will have consumed CLEAR (B) before OP2 can use it, forcing a new operator to be added to the plan to produce a new instance of CLEAR (B). Thus, the final solution will be the same as that returned by STRIPS.

Node 3



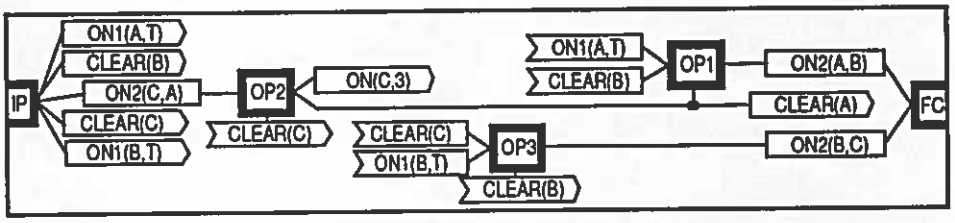
Our use of abstraction differs from the use of abstraction in ABSTRIPS. ABSTRIPS did not use a hierarchy of operators, and did not specialize operators in the sense used in OPIE. To model ABSTRIPS, we use a domain-specific criticality strategy for deciding which unproduced persistence to address. High criticality persistences are addressed before low criticality persistences.

Although Sacerdoti described ABSTRIPS as using abstraction, abstraction is not required to implement the notion of criticality. Everything that ABSTRIPS does can be done by OPIE using only the attribution refinement and a precomputed partial ordering on which persistences are to be attributed first. This strategy produces the same behavior as ABSTRIPS: delayed consideration of certain preconditions in favor of more critical preconditions.

As we mentioned earlier (p. 16), domain-specific heuristics for selecting which plan element to refine (e.g., which goal to attribute) can be combined with partial commitment to improve planner performance. Current work by Knoblock [89] is aimed at automatically computing precondition criticalities for a given set of operators. Like Sacerdoti, Knoblock uses 'abstraction levels' as a way of indicating criticality. OPIE does not compute criticalities; however, given the set of criticalities computed by Knoblock's ALPINE, OPIE could replicate ALPINE's problem solving behavior without using abstraction.

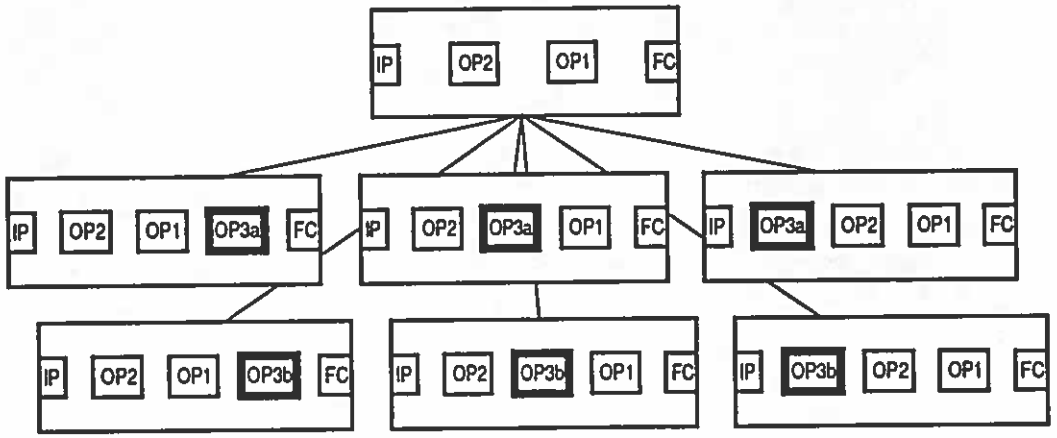
INTERPLAN [Tate 74] and **Waldinger's planner** [Waldinger 77] are like HACKER, except that a new operator can be inserted anywhere preceding the persistence being attributed. A separate node is created for each possible position of the operator in the plan. Unlike the previous planners, INTERPLAN and Waldinger's planner are complete and correct. If a solution to the problem exists, it will be contained in one of the nodes in the search space..

Node 4



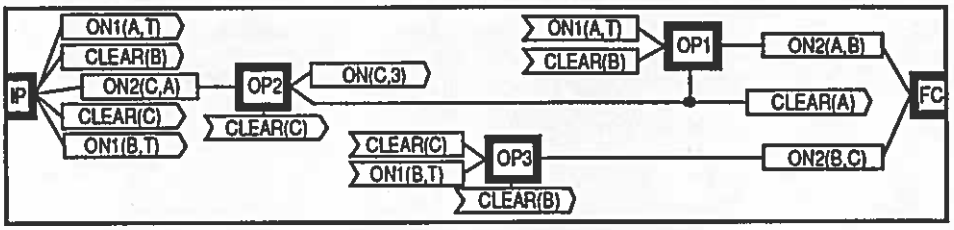
These planners are powerful enough to solve the Sussman Anomaly correctly, and, in fact, are powerful enough to solve any AI planning problem for which a solution exists.

Having resolved the completeness and correctness issues, the focus now turns to efficiency issues. The problem with trying every combination of operators in every possible ordering is that there are many such combinations. The figure below shows the three possible positions for operator 3 in the plan. Note that the branching factor is equal to the number of possible operators (in this case, 2) times the number of possible positions. The need for a more tractable approach for non-toy problems is apparent.

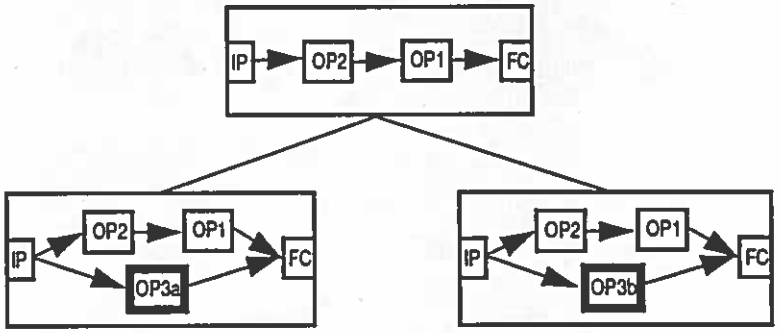


NOAH [Sacerdoti 77] eliminated the requirement that operators be totally ordered. Therefore, only one node was required to represent all potential positions of a new operator. NOAH delayed temporal commitments until enough constraints existed to determine the proper ordering. Node 4 shows the plan with OP3 unordered with respect to OP2 and OP1..

Node 4



NOAH avoids much of the unnecessary search required in INTERPLAN and Waldinger's planner because it does not commit to a particular ordering of operators. Thus, the branching factor is reduced to the number of alternative operators that can be used to produce a particular persistence. This is the best that a non-specializing, general purpose planner can do.



The type of hierarchical planning done by NOAH utilized a partonomic (i.e., part/whole) hierarchy but not a taxonomic operator taxonomy. For example, suppose OP2 and OP1 were combined as a macro operator which first clears a block and then moves it. This macro could be added to the plan in a single step instead of two steps.

Macro expansion is not treated as a separate plan refinement in OPIE. When a macro is inserted into a plan, it is immediately expanded. Because there is no choice involved, there is no reason to consider macro expansion a refinement of the plan.

TWEAK [Chapman 87] is a formalized version of NOAH; therefore, its solution to the Sussman Anomaly is the same as NOAH's solution. However, TWEAK explicitly searches in a space of plans, as does OPIE.

TWEAK uses a small number of plan modification operators. In OPIE, these modifications are subsumed by the attribution operation. *Establishment* is equivalent to simple attribution: unifying an unproduced persistence with an existing produced persistence. For example, attributing ON2 (C, A) to IP is an establishment. Both *step addition* and Chapman's *White Knight* correspond to OPIE's attribution with operator insertion: adding a new operator that produces an unproduced persistence. The use of persistences in OPIE makes the distinction between step addition and White Knight unnecessary. In TWEAK, the White Knight refinement is used in those cases where a relation holds early in the plan, but then is denied during the plan. Since two occurrences of the same relation are separate persistences, OPIE does not need a separate refinement for the two cases.

The rest of Chapman's modification operators are handled by constraints on plan elements in OPIE. *Promotion*, placing a user before a consumer, is enforced by the constraint that all users of a persistence must precede its consumer. For example, OP3 is constrained to precede OP1 because OP3 uses CLEAR (B) and OP1 consumes it. *Separation* is realized by object type constraints that prevent two persistences from unifying.

7.2 Specializing planners

NASL [McDermott 78] used explicit selection rules to choose among alternative methods for performing the same general task. These selection rules had to be written by the user. McDermott's *policies* served as a special class of heuristics that guided the specialization process. However, NASL contained no mechanisms for delaying the selection of a method so that constraints could be propagated from other decisions.

Friedland's MOLGEN planner [Friedland and Iwasaki 85] was the first planner to use an explicit *taxonomic* operator hierarchy. Friedland's work was based on observations of scientists designing molecular genetics experiments. The operator hierarchy was defined in advance by the user. Like NASL, Friedland's MOLGEN used domain-specific selection rules to choose among alternative specializations.

Friedland's planner retrieved complete "skeletal plans." It would specialize the first step in the plan to a primitive operator, then procede to the next operator in line. Friedland provided task specific rules to delay specializing one step if the outcome always depended on a decision later in the plan.

Stefik's MOLGEN planner [Stefik 80, Stefik 81a, Stefik 81b] also used an operator hierarchy defined by the user. Stefik provided a "refine-operator" mechanism, but did not describe exactly how the mechanism worked. In the detailed example Stefik discussed in his thesis ([Stefik 80]), MOLGEN apparently did not encounter any tasks that required choosing among alternative operators.

Stefik focused on choices involving the selection of objects to be used in the plan. MOLGEN would suspend any non-deterministic choices it encountered. If sufficient con-

straints were propagated from other decisions, the choice might become deterministic. If so, the choice would be re-activated and the single remaining option selected. If at any time in the planning process only suspended choices remained, MOLGEN would re-activate a suspended choice and make a guess. Domain-specific heuristics were used to determine which of the choices to re-activate.

Stefik mentions in a footnote ([Stefik 80, p. 49]) that, were "refine-operator" to encounter a non-deterministic choice, the choice would be suspended. However, the thesis does not say whether constraints could be propagated from other choices to reduce the number of alternative operators.

NASL, Friedland's MOLGEN, and Stefik's MOLGEN all depend to some extent on domain-specific rules. Therefore, it is not possible to predict exactly how they would solve the Sussman Anomaly. It is unlikely that either NASL or Friedland's MOLGEN would be able to solve the problem correctly without backtracking, regardless of the rules provided. Stefik's MOLGEN is the only one of the three able to suspend a choice because it is non-deterministic and then re-activate that choice if it becomes deterministic. Therefore, given the right selection rules and heuristics, Stefik's MOLGEN could have produced the same solution as OPIE under the LBFB strategy. The main differences are that OPIE generates the taxonomic operator hierarchy automatically, the user does not have to supply any selection rules, and the LBFB control heuristic is domain-independent.

Tenenberg [86] focused on the operator taxonomy, but did not actually implement a planner that used his ideas. Our implementation of operator and plan abstraction [Anderson and Farley 88] was greatly influenced by his notions.

Alterman's **PLEXUS** [Alterman 89] uses specialization for adaptive planning. Given a new problem and a solution to a similar problem, PLEXUS adapts the old solution to fit the new situation. This is accomplished by working through the steps of the solution in order from beginning to end. If the step can be applied as is, no adjustment is necessary. If the step is not appropriate in the new situation, PLEXUS finds an alternative by first moving upward in the taxonomic hierarchy to a more general operator that has the same effect. PLEXUS then specializes this operator to find a primitive operator that serves the same purpose as the original operator but can be applied in the current situation.

PLEXUS could not solve the Sussman Anomaly as stated because it is not designed to handle this type of problem. PLEXUS requires a solution to a similar problem as input. However, the specialization mechanism is very similar to that used in OPIE.

Partial commitment has also been used in constraint satisfaction problems [Mittal and Frayman 87]. That work is similar to ours in that a taxonomy of possible variable bindings is used to guide successive refinements.

7.3 Summary

STRIPS had two shortcomings that prevented it from being complete and correct. First, it did not attribute goals when new operators were inserted, allowing persistences to be consumed too early. This results in non-optimal plans. This shortcoming was corrected by HACKER. The second shortcoming was restricting where an operator could be placed in the plan, which resulted in incompleteness. INTERPLAN and Waldinger's planner corrected this and were shown to be complete and correct.

Subsequent planners addressed efficiency issues. NOAH was the first planner to use a form of partial commitment, allowing operators to be partially ordered. Stefik's MOLGEN allowed objects to be partially specified, and stressed the value of constraint propagation. OPIE adds a third form of partial commitment: to the selection of operators used in the plan.

8. Conclusion

All aspects of OPIE described in this paper have been implemented in ZetaLisp on a Symbolics ~~3600 series~~ Lisp Machine. OPIE's generalization module forms operator and object hierarchies, generating the necessary abstract operators and object types. The planner module uses the abstract operators to make partial commitments during planning.

We view partial commitment as a powerful planning tool. Unfortunately, it is not useful in all problems. In particular, certain classes of problems contain *constraint cycles*. That is, the best choice for decision A depends on the outcome of decision B, but the best choice for decision B depends on the outcome of decision A (possibly via a long chain of dependencies). Puzzles such as cryptarithmetic and crossword puzzles are designed to contain constraint cycles. Hamiltonian path problems such as the Traveling Salesman can be viewed as one giant constraint cycle, in which every decision depends on every other decision.

We suspect that many resource allocation problems and scheduling problems contain constraint cycles: a resource generally is either available or unavailable; conflicts cannot be resolved if a resource is *possibly* available. Partial commitment will not prevent the need to make guesses in problems containing constraint cycles (although it is still beneficial for choices not in a cycle).

On the positive side, there are many other problems that *do* benefit from partial commitment. We suspect that the vast majority of planning problems do not contain constraint cycles. In problems which are not intrinsically exponential, partial commitment offers the possibility of linear solutions.

We expect to find that partial commitment is most useful in rich domains with many interactions. In such domains, constraints can be quickly accumulated. Any mechanism that allows a decision maker to move past insufficiently constrained decisions to work on other parts of the problem increases the chances that the insufficiently constrained choices

will become fully constrained. We believe that planning with abstract operators as described in this paper represent one such mechanism.

Delaying commitment alone will not force decisions to become deterministic. Therefore, we are now looking at means for incorporating domain specific selection rules into the specialization process. We believe that the generalization hierarchy provides an excellent framework in which to index local rules that apply to a single choice point. We will compare choice-point-specific rules with the global production rules used in many expert systems.

Acknowledgments

We thank Josh Tenenberg, Steve Fickas, Bill Robinson, Keith Downing, Manuela Veloso and Craig Knoblock for helpful discussions and comments on earlier versions of this paper. The research reported here was supported in part by a National Science Foundation Graduate Fellowship.

References

[Alterman 88] R. Alterman, Adaptive planning, *Cognitive Science* 12, 393-421, 1988.

[Anderson and Farley 88] J. S. Anderson and A. Farley, Plan abstraction based on operator generalization, in *Proceedings of AAAI-88*, St. Paul, 100-104, 1988.

[Chapman 87] Chapman, D., Planning for conjunctive goals, in *Artificial Intelligence* 32, 1987.

[Fikes *et al.* 72] Fikes, Richard E., P.E. Hart and Nils J. Nilsson, Learning and executing generalized robot plans, in *Artificial Intelligence* 3, 251-288, 1972.

[Friedland and Iwasaki 85] Peter E. Friedland and Yumi Iwasaki., The concept and implementation of skeletal plans, Technical Report KSL 85-6, Stanford University, 1985.

[Korf 87] Korf, Richard E., Planning as search: A quantitative approach, *Artificial Intelligence* 33, 65-88, 1987.

[McDermott 78] McDermott, Drew V., Planning and acting, *Cognitive Science* 2, 1978.

[Mittal and Frayman 87] Sanjay Mittal and Felix Frayman, Making partial choices in constraint reasoning problems, in *Proceedings of AAAI-87*, Seattle, 1987.

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

[Sacerdoti 77] E. Sacerdoti, *A Structure for Plans and Behavior*, New York: American Elsevier, 1977.

[Stefik 81] Mark J. Stefik, Planning with Constraints (MOLGEN: Part 1), *Artificial Intelligence* **16**, 111-140, 1981.

[Sussman 75] Sussman, G., *A Computer Model of Skill Acquisition*, New York: American Elsevier, 1975.

[Tenenberg 86] Tenenberg, Josh, Planning with abstraction, in *Proceedings of AAAI-86*, Philadelphia, PA., 1986.

[Waldinger 77] Waldinger, R., Achieving several goals simultaneously, in Machine Intelligence 8, Elcock and Michie (eds), 1977.

[Wilkins 88] D. Wilkins, *Practical Planning*, Morgan Kaufmann, 1988.