Plan Abstraction Based On Operator Generalization

John S. Anderson
and
Arthur M. Farley

April 7, 1988
CIS-TR-88-05A*

Department of Computer and Information Science
University of Oregon

*This report is a revision of CIS-TR-88-05.

Plan Abstraction Based on Operator Generalization

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

Abstract

We describe a planning system which automatically creates abstract operators 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.

After the system has found a plan to achieve a particular goal, it replaces each primitive operator in the plan with one of its ancestors from the operator taxonomy. The resulting abstract plan is incorporated into the operator hierarchy as a new abstract operator, an *abstract-macro*. The next time the planner is faced with a similar task, it can specialize the abstract-macro into a suitable plan by again using the operator taxonomy, this time replacing the abstract operators with appropriate descendants.

1. Introduction

The time complexity of search using weak methods is exponential, which limits its use to relatively restricted problem domains. Searching in a hierarchy of abstraction spaces has been shown to significantly reduce the time complexity of problem solving [Korf, 87]. The classic abstract planner is ABSTRIPS [Sacerdoti, 74]. In ABSTRIPS, abstract operators are created by dropping certain preconditions from the primitive operators. The relative importance of each precondition is determined in advance by the programmer. By ignoring minor details while the major steps of the solution are being determined, only a few steps need to be found at each level of abstraction. The total search time is the sum rather than the product of these small searches [Minsky, 63].

A second approach to reducing search is to store plans. If a problem is encountered more than once, recording and storing the solution as a macro-operator eliminates the need to re-derive it. The oldest system for generating macro-operators is STRIPS with MACROPS [Fikes *et al.*, 72]. In that system, triangle tables are used to store sequences

of steps. Every sub-sequence of a plan is available to be used as a macro-operator. More recent research projects involving the discovery and use of macro-operators include Korf's work [Korf, 85] and Soar [Laird *et al.*, 86]. Unfortunately, macro-operators built from sequences of primitive steps can only be used in a limited number of situations.

Hierarchical planners take advantage of both of these kinds of search reduction. Our approach is most similar to that of Friedland, whose MOLGEN planner [Friedland and Iwasaki, 85] uses *skeletal plans* which are like abstract macro-operators. For a skeletal plan to be executed, each abstract step must be replaced by a primitive operator. This is accomplished by traversing downward in an operator taxonomy from the abstract operator to a descendant which is executable, given some set of initial conditions or constraints. In MOLGEN, the operator taxonomy and the skeletal plans are provided by the programmer rather than being generated by the planning system. In this paper, we describe a system that generates an operator taxonomy as well as generating and using skeletal plans.

Tenenberg [86] proposes using an operator taxonomy to guide the creation of *plan graphs*, which are abstract versions of triangle tables. A plan graph consists of a sequence of primitive operators and a portion of each operator's family tree. A plan graph might be viewed as a skeletal plan together with one of its specializations and all of the abstract operators in between. As Tenenberg's system was not implemented, we do not know how a plan graph would actually be used in planning.

This paper describes PLANEREUS, a planning system which builds up its own hierachically-structured knowledge base. The major contributions of our work are the definitions of automatic means for creating abstract operators, forming operator and object taxonomies, and generating abstract plans. Our goal is to create abstract macro-operators to be used by a hierarchical planner.

The next section of the paper describes two techniques used in generating abstract operators. One requires creating abstract object types, each specifying a class of objects which may fill a particular role in an abstract operation. The formation of the operator and object hierarchies is discussed in Section 3. Once the operator hierarchy has been formed, hierarchical planning techniques are used to find a sequence of primitive steps to achieve a particular goal. Section 4 explains how the operator taxonomy is then used to create *abstract-macros* from primitive plans. The final section discusses future work, including the use of plan abstraction in analogical problem solving.

The terminology used in this paper is generally consistent with previous descriptions of STRIPS-style planners [Nilsson, 80]. An operator contains an *add list, delete list*, and *precondition list*. We will refer to these three lists together as the *relation lists* of the operator. In addition, we include a fourth list, the *object list*, which specifies the types of the objects that participate in the operation. For example, a typical blocks-world operator is defined as follows:

```
        operator   PUT-BLOCK-ON-TABLE
     description   "put a block on the table"
         objects   ROBOT(?R), BLOCK(?B), TABLE(?T)
   preconditions   GRASPED(?B,?R), CARRIED(?B,?R), NEAR(?R,?T)
         deletes   CARRIED(?B,?R), GRASPED(?B,?R)
            adds   ON(?B,?T)
```

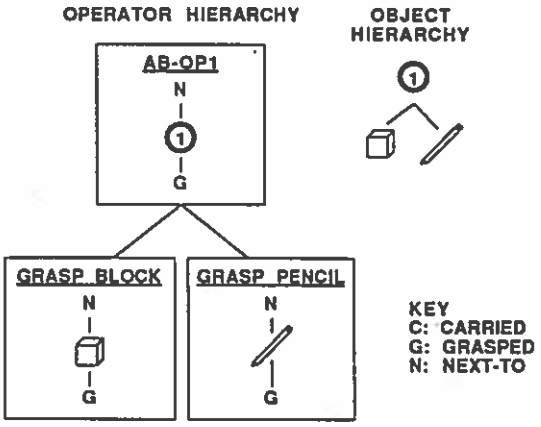

Figure 1. Generalizing on object types.

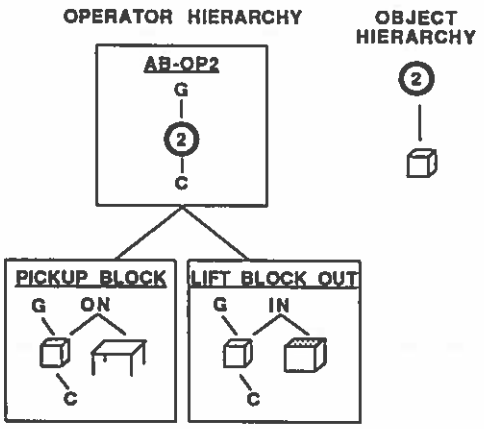

Figure 2. Generalizing on relations.

2. Operator Generalization

The first step in our approach to abstract planning is creating abstract operators 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 literals on their respective relation 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 operators as the leaves and abstract operators as the internal nodes.

We use two forms of generalization, which may occur separately or together during the construction of an operator taxonomy (as described in Section 3). The first form of generalization occurs when two operators have equivalent relations but different object types. For example, the operators GRASP-BLOCK and GRASP-PENCIL both require the relation NEXT-TO (ROBOT, ?X) and add the relation GRASPED (ROBOT, ?X). However, in one case the object type of ?X is BLOCK and in the other case it is PENCIL. PLANEREUS generalizes the operators by forming both an abstract operator and an abstract object type.

Figure 1 presents an example of generalization over object types. The operators are depicted by the large squares. An arc between two operators indicates a parent-child relationship. Inside the operators, icons signify the object variables. Above and below the icons are letters indicating relations, with arcs drawn to their arguments.[1] The relations above the icons indicate the preconditions of the operator; those below the icons are produced by the operator. Next to the operator hierarchy is the object hierarchy, which is formed in conjunction with the operator hierarchy.

Like its children, AB-OP1 requires NEXT-TO (ROBOT, ?X) and produces GRASPED (ROBOT, ?X). However, AB-OP1 has neither PENCIL (?X) nor BLOCK (?X) on its object list. Instead, it has AB-OBJ1 (?X), an abstract object type which includes, at this point, pencils and blocks. If another GRASP operator, such as GRASP-BALL, were added

[1]To simplify the diagrams, the ROBOT object is not shown and the relations involving ROBOT are drawn with only one argument.

```
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,
   INSERT (NEW, S).

Define INSERT (parameters NEW, S)
  For each operator OP in S, GENERALIZE (NEW, OP).

Define GENERALIZE (parameters NEW, OP)
  If NEW is a specialization of OP,
   Then LINK-PARENT-CHILD (OP, NEW).
   ElseIf NEW is a generalization of OP,
    Then LINK-PARENT-CHILD (NEW, OP) and
         INSERT (NEW, parents of OP).
    ElseIf NEW and OP share relations,
     Then create TMP with their common relations,
          If TMP matches existing abstract operator AB,
           Then LINK-PARENT-CHILD (AB, NEW), discard TMP.
           Else LINK-PARENT-CHILD (TMP, NEW),
                LINK-PARENT-CHILD (TMP, OP), and
                INSERT (TMP, parents of OP).
     Else {New and OP have no shared relations}.
```

Figure 3. Algorithm for adding operators to the operator hierarchy.

to the operator set, the new operator would be made a child of AB-OP1 and the object type BALL would become a child of AB-OBJ1. Thus, AB-OP1 is an abstract operator for "grasping" and AB-OBJ1 represents "grasp-able" objects.

The second type of operator generalization occurs when two operators share only some of their relations. For example, consider the operators PICKUP-BLOCK, for getting a block from the table, and LIFT-BLOCK-OUT, for getting a block from a box (Figure 2). Both add the relation CARRIED (?X, ROBOT) and have the precondition GRASPED (?X, ROBOT). However, PICKUP-BLOCK deletes the literal ON (?X, ?Y), while LIFT-BLOCK-OUT deletes IN (?X, ?Y). Generalizing these two operators gives us AB-OP2 which contains their common relations. Notice that only those object variables mentioned in the shared literals are generalized and included in the abstract operator.

3. Operator and Object Taxonomies

In planning by means-ends analysis (MEA), an operator is selected for consideration if a literal from the goal description matches a literal on the operator's add list. For an abstract operator to be useful to an MEA planner, it must add at least one literal. Therefore, PLANEREUS generates sub-hierarchies consisting of operators that share one or more literals on their add lists. The algorithm for adding new operators to the hierarchy is given in Figure 3. The major points are illustrated in the following example

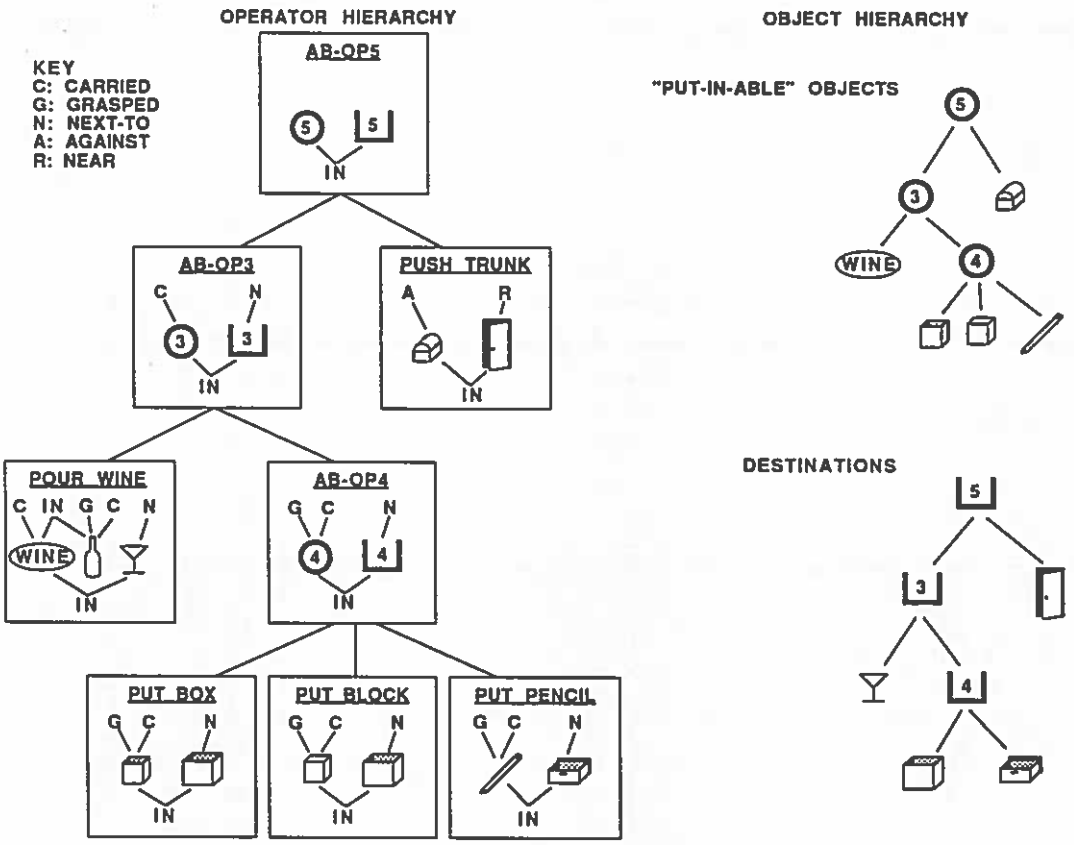

Figure 4. The "put-in" operator sub-hierarchy and related object sub-hierarchies.

of how one sub-hierarchy, shown in Figure 4, is constructed.

Suppose the operator PUT-BLOCK, for placing a block into a box, is the first operator in the hierarchy. Next, suppose POUR-WINE, which transfers wine from a wine bottle to a glass, is added. The new operator is compared to the old, and the common literals are extracted to form the abstract operator AB-OP3. In addition, abstract object types are formed to represent the classes of "put-in-able" and "destination" objects.

The next operator to be added is PUT-PENCIL, for putting a pencil into a drawer. The new operator is compared to each of the primitive operators of the sub-hierarchy in turn. It is first compared to PUT-BLOCK. The two operators have equivalent relation lists but different object types. A new abstract operator, AB-OP4, and two abstract object types are formed. PUT-BLOCK and PUT-PENCIL are made children of AB-OP4, as shown in Figure 5.

Now AB-OP4 must be added to the hierarchy. Because it is an abstraction of PUT-BLOCK, it belongs somewhere above PUT-BLOCK in the hierarchy. Therefore, AB-OP4 is compared to AB-OP3, the parent of PUT-BLOCK. Since AB-OP4 contains all of the relations found in AB-OP3, plus others, it is a specialization of AB-OP3 and is placed between AB-OP3 and PUT-BLOCK. The direct link between AB-OP3 and PUT-BLOCK is deleted, since the indirect link through AB-OP4 replaces it. The LINK-PARENT-CHILD

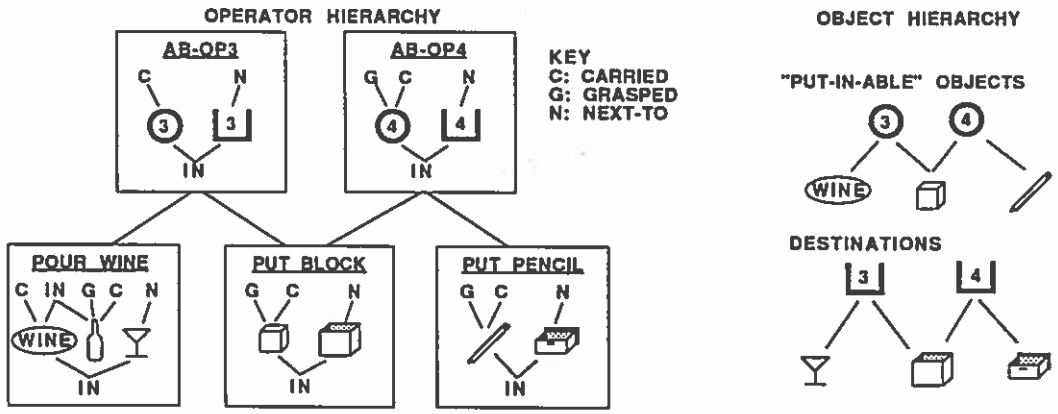

Figure 5. An intermediate state of the "put-in" sub-hierarchy.

procedure checks for and eliminates redundant links of this type. As a result, the order in which the primitive operators are added does not affect the final structure of the operator hierarchy.

Now PLANEREUS resumes consideration of PUT-PENCIL, comparing it to POUR-WINE. AB-OP3 generalizes PUT-PENCIL and POUR-WINE. Since AB-OP3 is already an ancestor of PUT-PENCIL no changes are made.

Next PUT-BOX, for putting one box into another, is made a child of AB-OP4. This single link reflects the generalization of PUT-BOX with POUR-WINE as well as with PUT-BLOCK and PUT-PENCIL. When compared to PUT-BLOCK, the common literals are exactly those in AB-OP4, so PUT-BOX is made a child of AB-OP4. When PUT-BOX is compared to POUR-WINE, their generalization is AB-OP3. Since PUT-BOX is already a descendant of AB-OP3, no additional changes are necessary.

The last operator to be added to this sub-hierarchy is PUSH-TRUNK, for moving a trunk into a room. When it is compared to PUT-BLOCK, AB-OP5 is formed. AB-OP5 is more general than either AB-OP4 or AB-OP3 so it is moved to the top of the sub-hierarchy, shown in Figure 4.

The sub-hierarchy shown in Figure 6 illustrates an additional point. An operator which contains two instances of the same literal can be matched with an operator containing one instance of the literal in two ways. For example, picking up a wine bottle causes both the bottle and the wine to be carried. In comparing PICKUP-BLOCK to PICKUP-WINE-BOTTLE, the block can correspond to either the bottle or the wine. In the former case, the common ancestor is AB-OP7, since both block and bottle are initially on the table and grasped. In the latter case, the common ancestor is AB-OP8. Both of these mappings are included in the operator hierarchy.

The object sub-hierarchies group object types into categories based on functional similarity. For example, the object type sub-hierarchies shown in Figure 4 collect "put-in-able" objects and containers which are "destinations" into separate categories. Since the same object type may appear in several sub-hierarchies, we are building functional definitions of the object types, based upon the roles they play in operators. For example, from the operators defined so far, a box is understood to be a pick-up-able, put-in-able

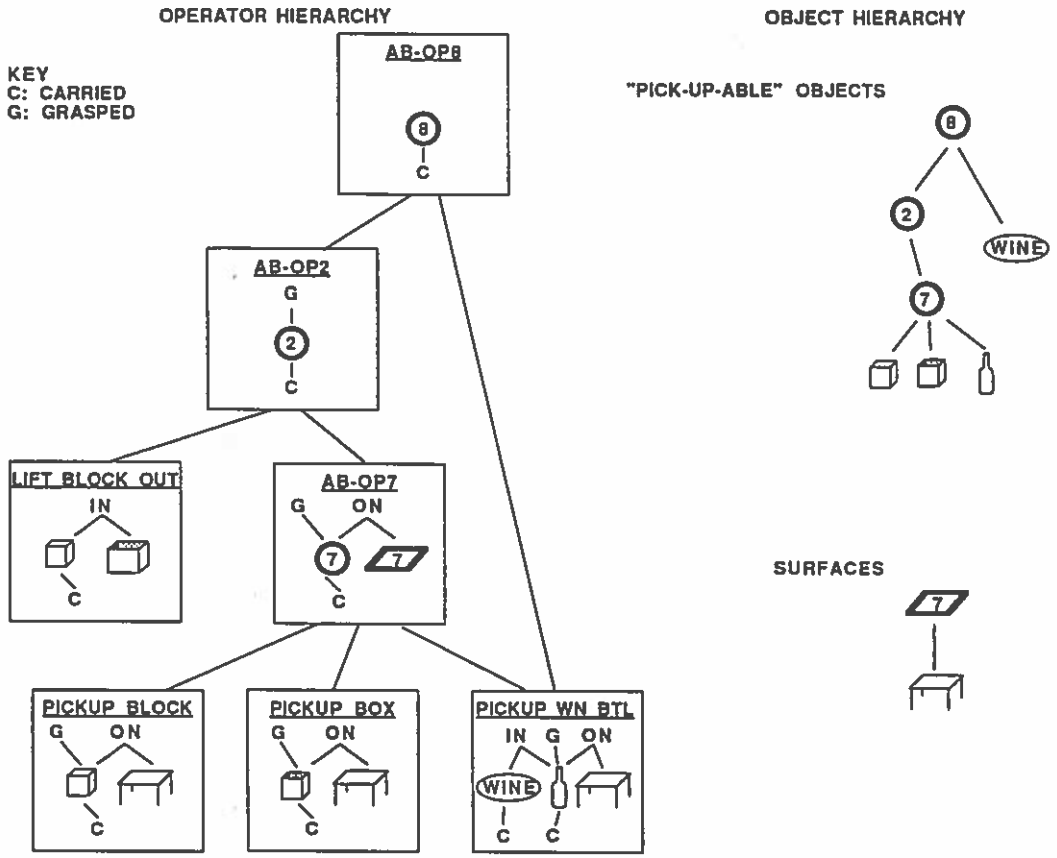

Figure 6. The "pick-up" operator sub-hierarchy and related object sub-hierarchies.

object which can serve as a source or destination container for other objects. (Note that all of the box icons in the figures correspond to the same node in the object hierarchy.)

4. Plan Abstraction

Establishing an operator hierarchy is a major step towards our goal of automatically generating abstract plans to be used by a hierarchical planner. While a primitive plan can solve a single problem, an abstract plan can be specialized in different ways to solve a variety of related problems [Friedland and Iwasaki 85]. PLANEREUS forms an abstract plan by replacing each operator in a primitive plan with one of the operator's ancestors from the taxonomic hierarchy. It then saves the sequence of abstract operations as an *abstract-macro*. The primitive plan is discarded, since it can be recreated by specializing the steps in the abstract-macro.

A key issue is determining which one of a primitive operator's ancestors should be used in the abstract plan. Our approach is to find the most general plan that retains the same *producer-user structure* as the original plan. An operator with a particular literal on its add list is said to be a *producer* of that literal, while an operator with the literal on its delete list is a *consumer* of the literal. An operator with the literal on its

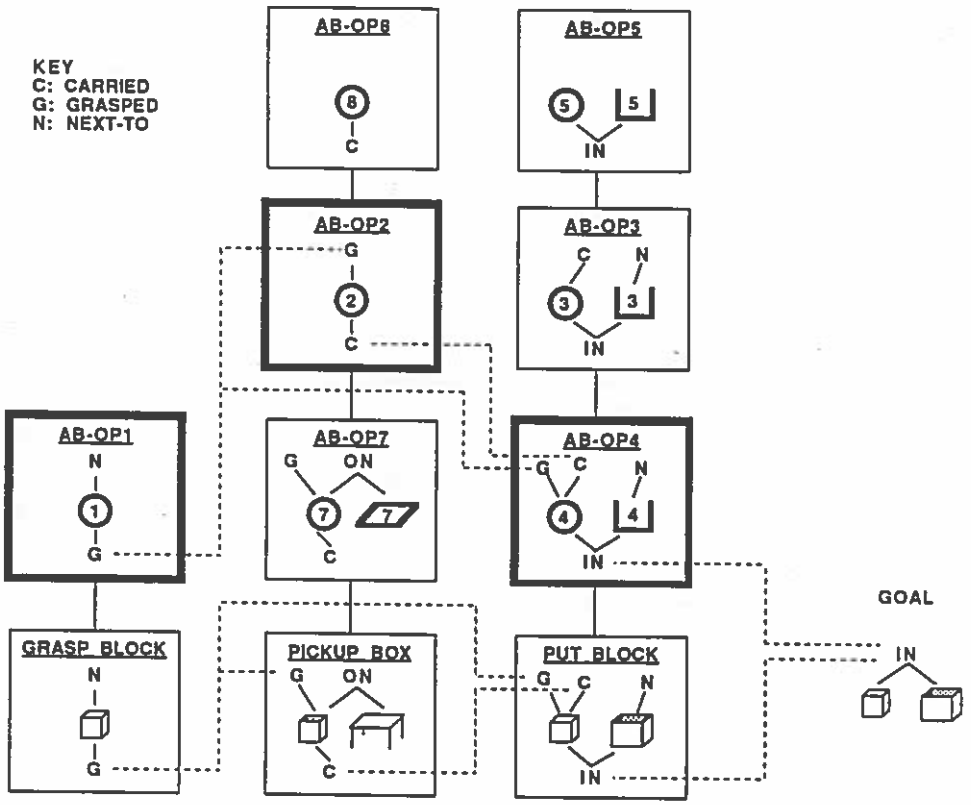

Figure 7. Abstracting a plan.

precondition list is a *user* of the literal, and is said to *require* the literal. In addition, a literal from the goal description is *required* (used) by the goal. Thus, each literal that is produced by an operator and later used by either an operator or the goal can be seen as a link between the producer and the user. These links determine the producer-user structure of the plan. The producer-user links reflect the *purpose* of each step of the plan, since an operator is included in the plan only if it produces a literal required by the goal or another operator.

We generate an abstract plan from the primitive plan as follows. First, we mark those literals in the add and precondition lists of the primitive operators that contribute to the producer-user structure of the plan. Then, for each primitive operator, we move upward in the taxonomy, examining the precondition and add lists of each ancestor in turn. Finally, we select the highest operator in the sub-hierarchy which contains all of the marked literals in its respective relation lists.

In Figure 7, the bottom row of primitive operators are the steps in a plan for putting a block into a box. At the right is the goal of the plan: IN (?BLOCK, ?BOX). The producer-user structure of the plan is indicated by the dashed lines connecting the place a literal is produced with the place(s) it is used. Above each primitive step are its ancestors from the operator hierarchy. According to our scheme, the operators selected for inclusion in the abstract plan are AB-OP1, AB-OP2 and AB-OP4, shown in bold outline. The abstract plan retains the producer-user structure of the original plan.

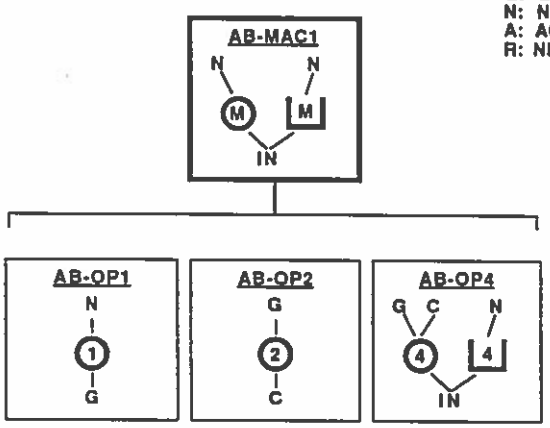

Figure 8. Forming an abstract-macro.

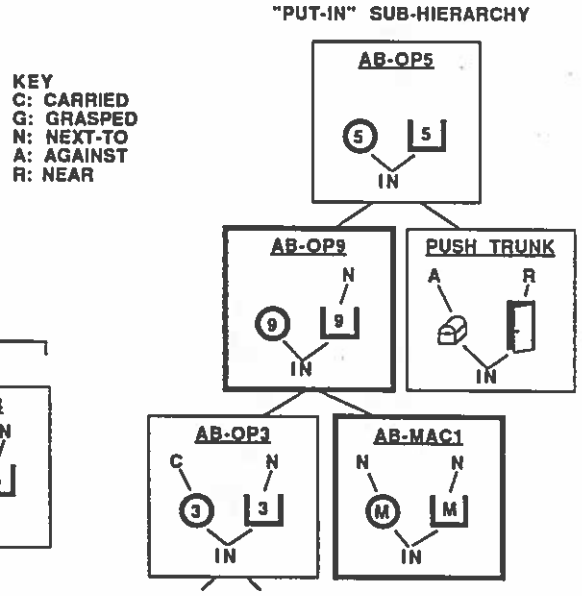

Figure 9. Inserting the abstract-macro.

Note that the operator selected is not always the most abstract ancestor of a primitive step. Selecting a more general operator can lead to a plan in which some steps serve no purpose. For example, if AB-OP8 and AB-OP3 were used instead of AB-OP2 and AB-OP4, GRASPED (?X, ROBOT) would no longer be required by the plan and the first step would have no connection to the final goal. If AB-OP5 were used, neither of the first two steps would be linked to the goal. We would then be losing the knowledge gained during the planning process.

Once the abstract operators have been selected, they are put together into an abstract-macro which has relation and object lists like the other operators. Figure 8 presents AB-MAC1, the abstract-macro created from the plan in Figure 7. The preconditions of each step are regressed to the front of the sequence; those not produced within the plan itself become the preconditions of the abstract-macro. Similarly, the adds not consumed and deletes not produced in the plan become the adds and deletes of the abstract-macro.

Forming the object list of the abstract-macro may require forming new abstract objects which span several sub-hierarchies. For example, AB-MAC1 involves an object which must be grasp-able, pick-up-able and put-in-able.

Finally, the abstract-macro is added to the operator hierarchy using the same process that was used for the primitive operators. AB-MAC1 would be added to the "put in" sub-hierarchy as shown in Figure 9. A new abstract operator, AB-OP9, is created in the process. The part-whole relationship between AB-OP4 and AB-MAC1 does not affect their relative positions in the hierarchy (cf. Figure 4).

A useful feature of abstract-macros is that new operators added to the system are automatically incorporated into existing abstract plans. An operator that becomes a descendant of AB-OP2, for instance, automatically becomes a potential step in a specialization of AB-MAC1.

5. Conclusion and Future Work

All aspects of PLANEREUS described in this paper have been implemented. PLANEREUS can form operator and object hierarchies, generating the necessary abstract operators and object types. It can also form abstract-macros from primitive plans. The planner module has been designed and partially implemented. Future work will include running benchmark tests to compare the performance of the planner given different amounts of knowledge (i.e., primitive operators alone; operator taxonomy alone; operator taxonomy with abstract-macros).

Like triangle tables [Fikes *et al.*, 72], abstract-macros contain enough information to use any sub-sequence of steps as a macro. Unfortunately, increasing the number of macros also increases the amount of search required to find the right one. One solution would be to store abstract plans for only certain types of problems. For example, [Korf, 85] discusses problems with *non-serializable sub-goals* as one class for which macros are useful. Another approach is used in Soar [Laird *et al.*, 86], which improves the efficiency of its macro-operator representations by noting common sub-sequences.

The purpose of forming abstract-macros is to allow a solution for one task to guide the construction of a solution for a similar task. Thus, PLANEREUS can be considered an analogical problem solver [Carbonell, 86]. Because of the methods we employ for generalizing operators, similarity between tasks is a function of relations in common rather than objects in common [Gentner, 83]. A limitation of the current system is that it only matches relations with identical predicate names when generalizing operators. Thus, OVER (?X, ?Y) does not match ABOVE (?X, ?Y). It would be useful to build up a relation hierarchy in the same way we are building the operator and object hierarchies.

An important next step is to determine to what extent the methods described here can be used to transfer planning knowledge between what would usually be considered separate task domains. Note that a single operator sub-hierarchy can span several domains. For instance, the "put-in" sub-hierarchy might span blocks-world, programming (assignment), and cooking domains. We will investigate the formation, selection, and specialization of abstract-macros that cross task domain boundaries. These processes constitute the *derivation* of the new plan [Carbonell, 86], and are a critical part of plan reuse and analogical problem solving. We are working towards an approach to problem solving that is predominantly top-down refinement rather than backward-directed search.

Acknowledgements

We thank Steve Fickas, David Novick, Bill Robinson, Keith Downing, and Ken Blaha 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

Carbonell, Jaime G. 1986. "Derivational analogy: A theory of reconstructive problem solving and expertise acquisition," in R.S. Michalski *et al.* (eds.), *Machine Learning Vol. II*, Morgan Kaufmann, Los Altos, CA.

Fikes, Richard E., P.E. Hart and Nils J. Nilsson. 1972. "Learning and executing generalized robot plans," in *Artificial Intelligence* **3**, 251–288.

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

Gentner, Dedre. 1983. "Structure-mapping: A theoretical framework for analogy," in *Cognitive Science* **7**, p155.

Korf, Richard E. 1985. "Macro-operators: A weak method for learning," in *Artificial Intelligence* **26**, 35–77.

Korf, Richard E. 1987. "Planning as search: A quantitative approach," in *Artificial Intelligence* **33**, 65–88.

Laird, John E., Paul S. Rosenbloom and Allen Newell. 1986. "Chunking in Soar: The anatomy of a general learning mechanism," in *Machine Learning* **1**, 11-46.

Minsky, Marvin. 1963. "Steps towards artificial intelligence," in E.A. Feigenbaum and J. Feldman (eds.), *Computers and Thought*, McGraw-Hill, New York.

Nilsson, Nils J. 1980. *Principles of Artificial Intelligence*, Tioga, Palo Alto, CA.

Sacerdoti, Earl D. 1974. "Planning in a hierarchy of abstraction spaces," in *Artificial Intelligence* **5**, 115–135.

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