The GRAPPLE Plan Formalism

Karen E. Huff and Victor R. Lesser

COINS Technical Report 87-08

April, 1987

Computer and Information Science Department
University of Massachusetts
Amherst, MA. 01003

This work was supported in part by the Air Force Systems Command, Rome Air Development Center, Griffiss Air Force Base, New York 13441-5700, and the Air Force Office of Scientific Research, Bolling AFB, DC 20332 under contract No. F30602-C-0008. This contract supports the Northeast Artificial Intelligence Consortium (NAIC).

Table of Contents

List of Figures and Tables

1.0 Introduction

The GRAPPLE plan formalism was designed to support the central paradigm in the implementation of an intelligent assistant. That paradigm involves performing both plan recognition and planning for a user working in a computer-based, professional domain. Two examples of these types of domains are software development and the automated office. Using a planning paradigm, the intelligent assistant can provide such help as:

- maintaining agendas (by enumerating the states yet to be satisfied in a plan),
- detecting errors (such as when a new user action cannot be recognized or violates a protected condition),
- correcting errors (for example, by informing of the need to satisfy a missing precondition or substituting the nearest expected action instead or suggesting that another action be performed first),
- answering user questions (which are interpreted as queries on either the state of the domain or the state of the plan), and
- automatically executing user tasks (by performing planning and execution monitoring).

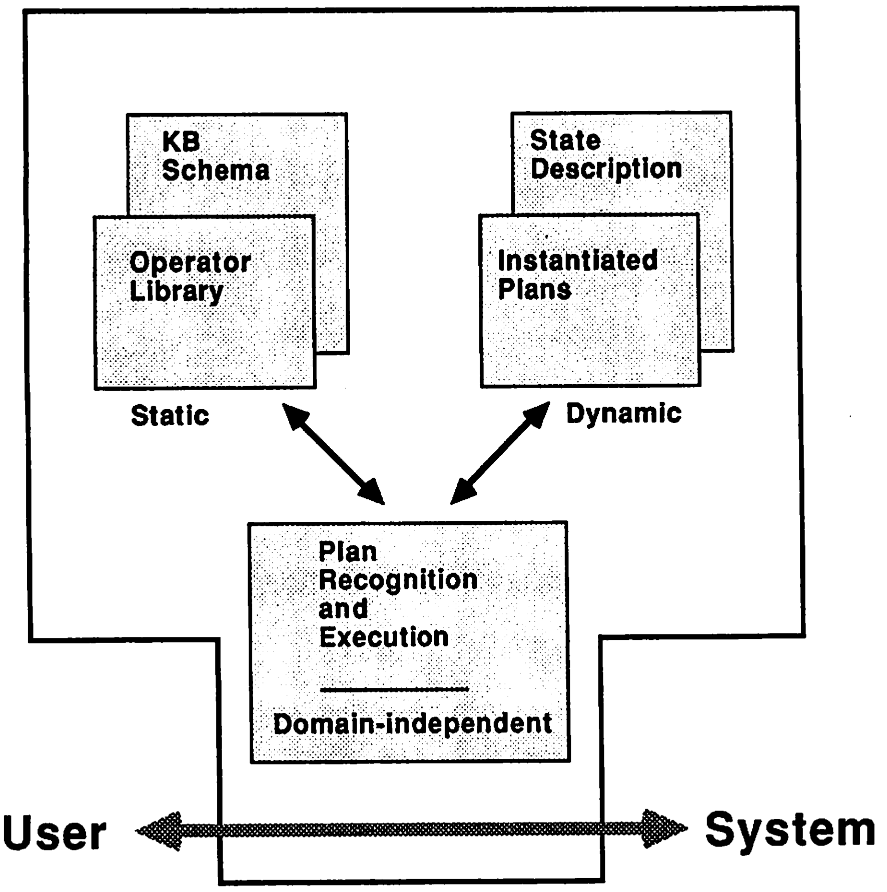
A schematic architecture for such an assistant is given in Figure 1. The assistant itself is domain-independent. Its domain knowledge is embodied in a set of operators (written in the GRAPPLE Plan Formalism GPF) which describe the actions possible in the domain. Using these operator definitions, complete plans can be constructed to explain a series of user actions (plan recognition) or to achieve a desired user goal (planning).

The GRAPPLE project is described in [2]. It has evolved from an earlier effort called POISE, described in [1,3,5]. The application of POISE to office automation tasks is discussed in [6]; its application to the software development environment is described in [8]. The results of the POISE project demonstrated the viability of a planning approach to intelligent assistance, and GRAPPLE is intended to build on and extend those results.

1.1 Requirements

In this section, we discuss the requirements that an intelligent assistant application places upon a plan formalism. We also mention the rationale for choosing an underlying formalism that is different from that used in POISE.

Figure 1: Architecture for An Intelligent Assistant



In the domains of interest, there are three distinct parties with distinct capabilities. The first party is a dumb agent (in this case, a computer system), capable of carrying out actions from a specific repertoire on explicit command, but without any facility for judging the global sense or advisability of those actions. The second party is a human user directing the agent, capable of planning and understanding the actions, but fallible; the fallibility arises from the complexity of the world state coupled with the complexity of the actions themselves. The final party is the intelligent assistant, with incomplete knowledge of the domain, but with the ability to operate accurately within that part of the domain where its knowledge is complete. Thus the intelligent assistant compensates for the fallibility of the user, but cannot entirely replace the user. To the user, it appears that the intelligent assistant augments the facilities of the dumb agent, in effect giving the dumb agent an acceptable level of "smarts".

A unique requirement derives from the fact that it is too complex to build a fully autonomous, automated agent to replace the user. For these domains, it simply is not possible to codify all the knowledge necessary to make the intelligent assistant as knowledgeable as the user -- what the expert user knows is not well-enough understood. Therefore, the plan formalism must allow the definition of incomplete plans, which cannot be executed without cooperative input from the user. The formalism must make a distinction between decisions which the intelligent assistant can make independently and those for which the assistant must have recourse to the user. This aspect of GRAPPLE plans represents a departure from previous planning work, where the planner has all the knowledge needed to be fully autonomous.

Another set of requirements stems from the fact that the human user remains in the picture, unlike the usual situation where the user is simply replaced by an automated agent. The intelligent assistant must be able to converse with the user, especially to answer user questions. GRAPPLE plans should reflect the user's view of the hierarchical levels of activities. Thus, use of hierarchical plans is motivated by a desire to have consistency between the assistant's and the user's pictures of the relationships between activities, independent of the traditional motivation of controlling search space to make planning more efficient. A related concern has to do with answering questions posed by the user about the world state. We want the world description to encompass the user's interpretation, in addition to the simple facts about the physical reality of the world. Thus, the world description in GPF will be richer than that of traditional planning systems. It will include not just the bare facts (block A rests on block B which rests on block C), but the interpretations placed upon those facts by the user (blocks A, B, and C form a structure which is some kind of column).

Additional requirements stem from the fact that even small applications in our target domains require the operator definition language to be engineered for real-world situations. This includes requirements for operators with large numbers of variables, repeated actions, complex constraints on operator variables, and "underconstrained" variables. Further, operators must be able to create new world objects (when a programmer uses the editor to create a new file, the effect on the world state is to create a new object of type file); such a feature is not commonly implemented in operator definition languages.

Finally, the plan formalism must support plan recognition as well as planning. Occasionally, these two applications require different sorts of information, and these differences must be accommodated. However, in both cases, the execution of plans is monitored by the intelligent assistant, so an interface is needed by which information on success or failure can be acquired. Such an interface has further uses: performing actions can lead to the acquisition of new information about the world (not just changed world states). One interface can serve as the conduit between the real world and its description within the intelligent assistant.

We wanted to act upon the insight gained from our work with the POISE system: namely, that more knowledge was needed in operator definitions. In particular, additional knowledge was required to deal with exceptional situations arising from multiple top-level goals being achieved in parallel. POISE could handle multiple, concurrent top-level plans, and that capability had to be preserved. But, POISE had insufficient information to reason about interactions among on-going plans, at any hierarchical level. Take a POISE plan of the form $A <- B\ C\ (D\ /\ E)$ which is read as "plan A consists of performing action B, then action C, then either actions D or E". We needed to add information to know when C might be redundant, because its goal had been already been achieved by other on-going legal actions; or, when C could not directly follow B, because certain effects of B were subsequently wiped out prior to C starting; or, what to do if C failed -- perhaps D is provided for the C-succeeding case, and E for the C-failing case. This last issue requires that we be able to decide whether C has in fact failed, which POISE had insufficient information to do.

We also knew from the POISE experience that the more complex a domain is, the more important it is to consider the difficulties of providing a complete operator library. It is important that the library be modular, so that one can add new operators without having to rewrite existing operators.

As a result of all these requirements, the GRAPPLE plan formalism has all the characteristics of a full-featured plan formalism suitable for the standard planning algorithms of the literature. It should be noted that traditional planning techniques cannot be transferred directly to this application due to many factors including:

- multiple top-level plans can be executing concurrently, implying that plan interactions cannot be reliably predicted because new top-level plans can start at any time,

- the planner is not fully autonomous (it lacks complete knowledge, as explained above),
- the goal of plan recognition is to recognize the user's *actual* plan, which may be different from an *optimal* plan,
- the goal of plan recognition is to recognize the user's actual plan *as it is being performed*, not after it is complete, so as to maximize the opportunity for detecting errors while they are still readily correctable.

1.2 Guide to Organization of this Report

The remainder of this report is divided into three parts. In Section Two, we give a precise definition of the GRAPPLE Plan Formalism GPF, the language in which operators are defined; the GRAPPLE semantic database, which is not separable from GPF, is also defined. In Section Three, we consider useful extensions which could be made to GPF. In Section Four (the final part), we summarize how the requirements for GPF were met and compare GRAPPLE to other plan formalisms.

A formal grammar for GPF appears in Appendix A. An operator library for a single, complete example domain appears in Appendix B. A companion technical report [9] describes how GPF is used to model the software development domain; an extensive set of software development operators written in GPF is given there.

It is traditional that no plan formalism may be properly introduced to the field without a blocks-world example; therefore, we follow tradition and present various plans for stacking and unstacking blocks. The simplest of these plans have appeared in the literature many times, and thus serve as a straightforward way for the reader to compare GRAPPLE to other plan formalisms. We also present more complex blocks-world examples which serve to show the strengths of GPF for complex domain modeling.

The blocks world was originally conceived as a robot problem. For the application of an intelligent assistant, it is more appropriate to think in terms of a small child playing with a robot manipulating blocks, while an adult looks on. The intelligent assistant acts the role of the adult: interpreting the child's commands to the robot as attempts to build meaningful structures (thus, performing plan recognition), or demonstrating to the child how to build meaningful structures (thus, planning and executing the necessary primitive actions).

2.0 Formal Definition of GPF

2.1 Overview of Operator Definitions

GRAPPLE provides for the hierarchical definition of operators. An aggregation hierarchy is used, where multiple, lower-level operators are aggregated into a single higher-level operator. Each lower level operator is a part of the higher level operator. (This is in distinction to a generalization hierarchy, where lower level operators are specializations of higher level operators.) At the lowest levels of the hierarchy, we have *primitive* operators which correspond to the atomic actions in the domain. Operators at all other levels are *complex* operators, defining activities at higher levels of abstraction.

GRAPPLE operators are fundamentally state-based. They follow the state transition approach introduced with the earliest planning work. This is in contrast to event-based (also called behavioral) formalisms, of which POISE plans are an example. In an event-based system, the emphasis is on sequences of actions; in a state-based system, the emphasis is on sequences of states.

Every GPF operator includes clauses which define the goal, the precondition, and the effects of the operator. An example of a (partial) GPF operator showing just these clauses is given in Figure 2; this is the traditional example of the primitive operation to stack one block on another. (Throughout, we follow the convention that the operator template, including GPF reserved words, is given in uppercase, and the details of this specific operator appear in lowercase.) The interpretation of the basic GPF clauses is as follows:

- If the operator is executed in an initial state A in which the precondition is true, then the effects are realized causing a transition to state B.
- If the execution of the operator succeeded, then the goal will be true in state B; otherwise, it will be false. (The Figure 2 example does not yet deal with failure -- we will expand the example later to show how it is handled.)

While the goal, precondition and effects clauses are the core of a operator definition, there are other operator clauses. All operators have a constraints clause which describes relationships among operator variables. If the operator is primitive, it has an observe clause, which is part of the interface to the real world necessary for both plan recognition and plan execution. Complex operators have a decomposition clause, which shows how the complex operator is

FIGURE 2: A Basic Operator

```
(OPERATOR stack IS-PRIMITIVE
; This is an operator for moving a single block on top of another;
; the block to be moved must be available directly to the robot arm
; (i.e., must have no other blocks on top of it)
; and must also be on the table.
; The block which is to serve as the base must have its top clear
; to receive the block being moved.

(GOAL              on(x,y) )

(PRECOND           (clear(y) AND clear(x) AND ontable(x)) )

(EFFECTS           (ADD on(x,y))
                   (DELETE clear(y))
                   (DELETE ontable(x)) ) )
```

broken down into simpler parts. That completes the overview of operator clauses.

As would be expected in a state-based formalism, there is a database which is used to describe the state of the domain world. Domain-specific predicates, functions and initial constants are predefined, and constitute the schema of the database. Queries on the state of the world are then expressed as formulas in first order predicate calculus. Since the database serves as both the state definition and the description of all domain objects, we call it a "semantic database" or SDB.

2.2 Major Operator Clauses

2.2.1 Goal Clause

The goal clause identifies the particular database state which is meant to be achieved by the execution of this operator. Obviously, this is a family of database states, since a typical goal will mention a small subset of the database predicates, leaving the truth value of other predicates unspecified; the specific member of the family is immaterial to the operator. The goal is a formula whose truth may be determined by querying the database. Other operators may have identical or similar goal clauses, in which case there are alternate ways to achieve this goal.

We make a distinction between the *goal* of a operator, and its *purpose*. While the goal is predefined and static, the purpose is decided dynamically when operators are instantiated and a plan hierarchy is constructed. The purpose of a operator consists of contributing to goal and/or precondition satisfaction for other parts of the plan hierarchy. If the plan hierarchy includes an operator P whose precondition is A, and if another operator Q whose goal is A appears in the hierarchy as part of the expansion of P, then the purpose of Q is to satisfy the precondition of P. There might be another plan hierarchy with an operator R, whose goal is A AND B, and Q might appear in the hierarchy as part of the expansion of R; then the purpose of Q would be to contribute to the satisfaction of the goal of R. In each case the purpose is different, but the goal of Q is always the same.

In some sense, the goal clause in GPF is redundant because the purpose of an operator is the real determiner of success or failure, and because an explicit effects clause is provided. The presence of the goal clause in GPF is intended to give the operator designer the opportunity to provide some focusing information. The goal clause lists the important effects of the operator, and thus distinguishes between the "main" effects of an operator and its "side" effects. The use of this information in constructing plan hierarchies is described in Section 2.6.4.

2.2.2 Effects Clause

The effects clause defines a state transition (from an initial state to a final state) which occurs as a result of executing this operator. If execution of the operator was successful, then the goal will be true in the final state of this transition. The effects clause is expressed as a set of atomic database operations such as making predicates true or false or creating new database objects; taken together as a single, uninterrupted database transaction, these atomic operations define the state transition. In the case of a complex operator, the state transition takes place **after** all subgoals are achieved.

The effects clause will usually include more than just those database operations which make the goal true. The effects clause is the means of updating the semantic database, including bare facts and interpretations of those facts; so, all knowledge to be gained from executing a particular operator should be described in the effects clause: goal-related "main effects" as well as "side-effects". We give a few examples:

- In the simple stack operator of Figure 2, the deletion of *clear(y)* and *ontable(x)* are side-effects of achieving *on(x,y)*. However, they cannot be omitted if we are to have an accurate state description for the state after a stack action for x and y.
- We might imagine that in the process of stacking (or unstacking a block), we would have access to information about the weight of the block; for the sake of argument, let us assume that this information is accessible only at this time. Then, we would want to record it in the semantic database, so that it was available later. For example, we might have other operations on blocks which were conditional on certain weight constraints.
- If we had operators to build structures (towers, bridges, fences, etc) out of the blocks at our disposal, we might want to record the color of a structure, based on the blocks of which it was composed. The color might be red, blue, ... or multi-color (for the case where mixed colors of blocks were used.) Recording the color of the structure might not be information needed by any other operator, but it might facilitate discussion with the user about the domain state.

2.2.2.1 Conditional Effects We allow the database operations of the effects clause to be conditional. Thus the effects clause actually defines a family of transitions; the goal of the operator will be true for some family members, and false for the rest.

Conditional effects can be used in two ways. First, we can define generic operations, and leave the fine distinctions to the effects clause. To do so, we make one or more database operations conditional upon some fact in the state prior to execution of the operator. This allows us to get extra mileage from a single operator: without conditional effects, we would have to make multiple operator definitions. Second, we can make the outcome of a operator dependent upon observations from the real world; this means that we have the ability to make the appropriate database updates both in the case of a operator succeeding as well as in the case of a operator failing. In this case, the database operation is conditional upon feedback from the real world. Examples of these two uses of conditional effects are as follows:

- Suppose we assume that when we stack one block x on another block y, x can come either from the table or from on top of another block z. Then, in the stack operator, the "side-effects" involving the new status of x will be either deleting *ontable(x)* or deleting *on(x,z)*. The choice of "side-effect" is conditional on the status of x prior to the stack action.

- Suppose the child's robot could not lift blocks whose weight was greater than a certain threshold, and suppose further that the only way to gauge the weight of a block is to attempt to stack or unstack it. Then it is possible that a stack operation would fail if the block were too heavy. In this case, either the effects as given in Figure 2 would be achieved (along with an effect recording that the weight of the block is below the threshold) or the blocks would remain in their original position and we would achieve a single effect: namely, recording that the weight of the block is above the threshold.

2.2.2.2 Effects of Complex Operators Although one might suppose that complex operators should not have effects clauses, we not only allow this, but believe that effects clauses for complex operators provide for more accurate domain modeling. Of course, if the goal of operator P was A AND B, and the decomposition of P was to achieve A and achieve B, then P itself would not strictly require any effects. When a complex operator does have an effects clause, it generally involves recording some higher-level semantic concept in the semantic database. For example,

- If we were building structures and had operators to paint blocks, then we might have a operator to make a structure red, consisting of painting all blocks in the structure red. The blocks are marked red (in the SDB) as each of the subgoals is accomplished, and the structure is marked red (in the SDB) as an effect of the paint-structure-red operator.

- If we are building a vertical structure with three blocks, then we can do so with two stack operations. The effects of the stack operations show x on y and z on x. As effects of the vertical-structure operator, we can introduce an object representing the structure, and show that blocks x, y and z are part of it. These effects give a higher-level semantic interpretation to the world state than is possible with just *on(x,y)* and *on(z,x)*.

2.2.3 Precondition Clause

The precondition clause establishes constraints on the initial state in which operator execution can start; these constraints must be met in order for the state transition defined in the effects clause to be valid. Another way to look at the precondition clause is to say that it defines an appropriate start state from which the goal may be achieved via this operator.

2.2.3.1 Use in Ordering Actions Operators must be executed in the order dictated by their preconditions. Ordering of operators is not specified in any other way; in particular, the temporal ordering rules found in an event-based formalism are not used. Preconditions allow an implicit concurrency among operators: if two complex operators have true preconditions, then both can be executing at the same time.

In the case of a complex operator, the precondition must be true **before** any subgoal is to be achieved. Therefore, it is good operator writing style to push preconditions down to the lowest possible operators in the hierarchy, so as to allow as much concurrency of higher-level tasks as makes sense for the domain. For example, we might have a set of operators including some to build structures of various types and others to take them apart. From such operators, we can define another operator to build a new structure out of the blocks of some existing structure. During execution of this re-use-blocks operator, we need not complete the dismantling prior to starting to re-build; if blocks are available in the right order, we can interleave dismantling with re-building. In this case, we do not want to have a precondition on re-building that requires that all necessary blocks be available. Availability should be a precondition on individual stacking operations, which satisfy the subgoals of re-building.

2.2.3.2 Types of Preconditions We make a distinction between two types of preconditions: *normal* and *static*. An operator may have both types, or only one type, or none. The precondition of a operator is understood to be the conjunction of the normal and static preconditions when both are present.

The normal precondition for an operator takes the form of a list of formulas, which are implicitly joined by the AND operation. Thus if the operator writer states:

```
(PRECOND    (A , B AND C) )
```

the complete normal precondition is understood to be the formula:

```
A AND B AND C
```

By dividing the entire normal precondition into separate parts, the operator writer is providing some heuristic information to the planning system about how to break the precondition into separately achievable parts. For the example above, the information indicates that A can be achieved separately but B and C can be achieved together. The use of this information is described in Section 2.6.4.

If a normal precondition is found to be false, it will be appropriate to take explicit steps to make it true. For static preconditions, this is not the case; a static precondition specifies universal conditions of applicability for the operator. A static precondition may involve some aspect of the database which is not changeable, or it may cover a case where it does not make sense (given the domain) to attempt to make the precondition true when it is false.

FIGURE 3: Static Precondition Examples

```
(OPERATOR stack IS-PRIMITIVE
; this is another way to move a single block on top of another.  Here we
;assume that blocks come in two types: those with flat top surfaces, and
;those with other types of top surfaces.  We add the static precondition
;that the base block must have a flat top in order for the other block to sit on it.

(GOAL               on(x,y) )

(PRECOND            (clear(y) , clear(x) , ontable(x) ))
                    (STATIC flattop(y)) )    ; here is the required condition

(EFFECTS            (ADD on(x,y))
                    (DELETE clear(y))
                    (DELETE ontable(x))  )


(OPERATOR unstack IS-PRIMITIVE
; this is the basic operator for taking one block off the top of another

(GOAL               ontable(x) )

(PRECOND            (clear(x))
                    (STATIC on(x,y)) )

(EFFECTS            (DELETE on(x,y))
                    (ADD clear(y))
                    (ADD ontable(x))  )
```

Two examples of static preconditions are given in Figure 3. The examples use an extended blocks-world example which includes blocks of two types: cubes and pyramids. No blocks can be stacked on top of a pyramid, because its top isn't flat. In the first Figure 3 operator, we see that the precondition requiring a flattop is a static precondition: the "flattop-ness" of a block is predetermined for a given block, since we are not dealing with a domain in which there are operators to turn cubes into pyramids, etc. In the second Figure 3 operator, we define the action of unstacking blocks. The precondition requires that the two blocks to be unstacked are presently stacked. If not, it doesn't make sense to stack them, since the operation unstack will simply undo that action. Hence, the precondition on the unstack operator is static.

Note: static preconditions involving unchangeable aspects of the world must be used carefully. They restrict the modularity of the operator library, in the sense that some operators are written in ways which are dependent on knowledge of what the other operators in the library do or don't do. They prevent the later, straightforward addition to the operator library of operators which do make this aspect of the database changeable. In the first Figure 3 operator, we would have to change the static precondition into a normal one if we decided to extend the world modeling to include the reshaping of blocks (as in a Lego system, where blocks are composable.)

2.3 The Semantic Database

A state description mechanism, which we have called the semantic database, is central to any style of state-based plans. In this section, we discuss how the semantic database can be informally modeled, and how it may be formalized from this model. We also discuss the issue of formalizing assumptions about the valid states represented in the semantic database.

2.3.1 Modeling the Semantic Database

We have found that the ER (entity-relationship) model of data [4] is a useful way to plan how to model the domain world. The ER model is appealing during the process of writing plans because it is highly intuitive, and it lends itself to an attractive graphical representation.

FIGURE 4: Blocks World

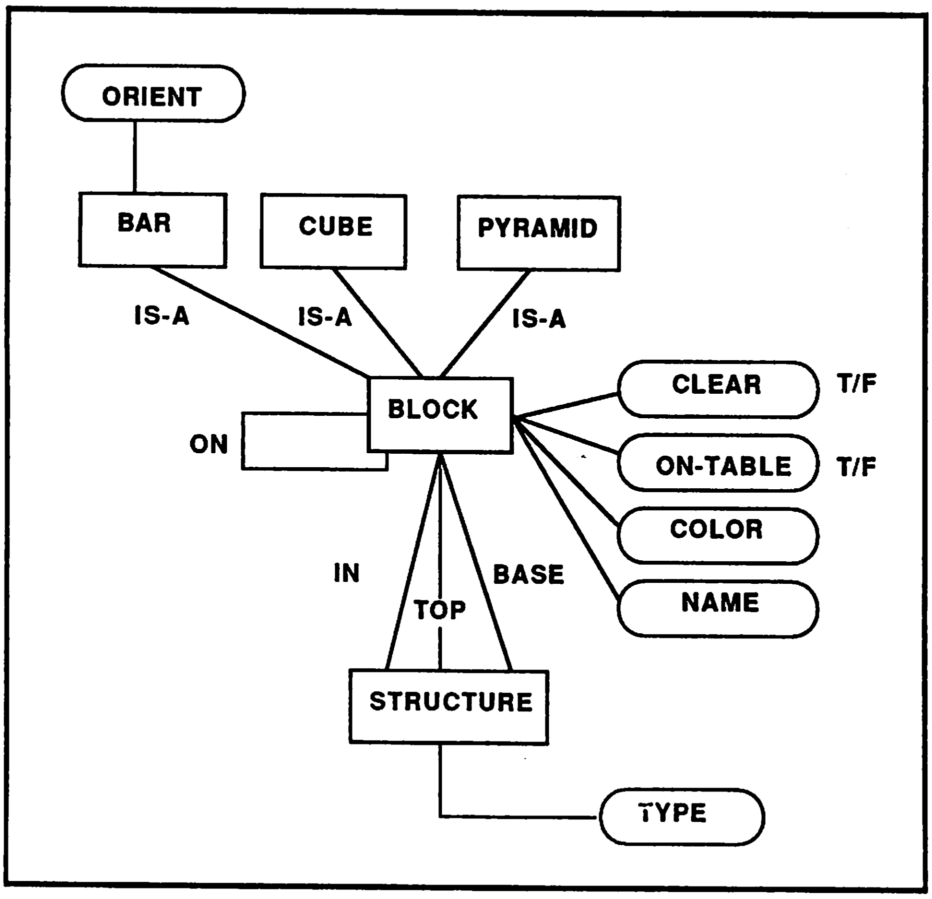


FIGURE 5: Semantic Database Formalized

Predicates (Extensional) attributes)

```
on(block, block)
ontable(block)
in(structure,block)
clear(block)
top(structure,block)
base(structure,block)
```

Predicates (Extensional, from ER

```
type-struct(structure, {unknown,tower,
        column ...})
type-block(block, {cube, pyramid, bar})
color(block, {red, green, blue})
        orient(bar, {horizontal, vertical})
name(block,string )
```

Functions

```
in-struct(block) : structure
            in-struct(x) = y IFF in(y,x)
top-of(structure): block
            top-of(x) = y IFF top(x,y)
base-of(structure): block
            base-of(x) = y IFF base(x,y)
```

Predicates (Intensional)

```
committed(b: block):
                    THEREEXISTS s |
                     in(s,b)
```

Constraints

```
IF in(s1,b) AND in(s2,b) THEN equal(s1,s2)
            ; "in" is many to one
IF on(b1,b2) AND on (b1,b3) THEN equal(b2,b3)
IF on(b1,b2) AND on(b3,b2) THEN equal(b1,b3)
            ;  "on" is one to one
IF equal(name(x),name(y)) THEN equal(x,y)
            ; block names must be unique
on-table(x) XOR THEREEXISTS y | on(x,y)       ;law of gravity
NOT on(x,x)        ; on is not reflexive
IF on(x,y) THEN (THEREEXISTS s | in(s,x) AND in(s,y))
            ; any stack of two or more blocks must be a structure
IF top(s,x) THEN clear(x)        ; what it means to be on top
NOT committed(x) IFF (clear(x) AND ontable(x))
            ;what it means not to be in a structure
IF top(s,x) THEN in(s,x)
            ; the top of a structure must be in the structure
IF base(s,x) THEN in(s,x)
            ; the base of a structure must be in the structure
IF base(s,x) THEN ontable(x)
            ; the base of a structure must rest on the table.
```

With ER, there are objects called entities, these entities have attributes, and the entities participate in relationships. Attributes are typed; we expect to handle strings, booleans, enumerations and numbers. As a further convenience in the ER schema, we allow the distinguished transitive relationship is-a between two entity types (as opposed to entity instances). This relationship is used to define a generalization hierarchy; it has the usual meaning that the one entity inherits all attributes and relationships defined on the other.

In Figure 4 we give an ER schema of a simple, but interesting blocks-world; in Figure 5, the predicate calculus formulation of this world is given (the translation from ER to predicate calculus is discussed in the next section). This example has been chosen to give the flavor of domain modeling which is appropriate for and possible with GPF plans. In this world, we have three types of blocks: cubes and pyramids and bars; they are shown in Figure 6. The blocks can be assembled into structures. (We will be limiting this world to vertical structures with tops and bases, in order to keep the examples to a manageable size.)

Since we will use this extended blocks world for all our remaining plan examples, we give in Figure 7 the primitive operations of this world. (A few operator features are used which have yet to be introduced.) Each stacking or unstacking action must now take the structure into account. This leads to two variations on stacking: start-struct (which introduces a new structure) and extend-struct (which builds on an existing structure). To keep the example simple, we assume that bars cannot be re-oriented (if they appear in the initial world in vertical position, so they stay, and similarly for horizontal position.) Examples of the kinds of structures which can be built using these primitive operations are given in Figure 8.

Figure 6: Extended Blocks World Blocks

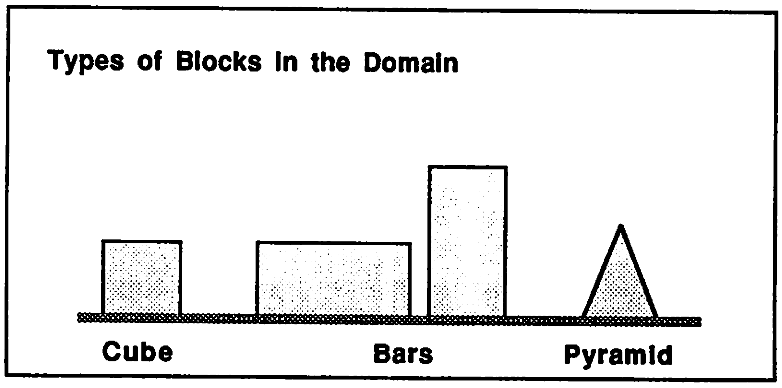
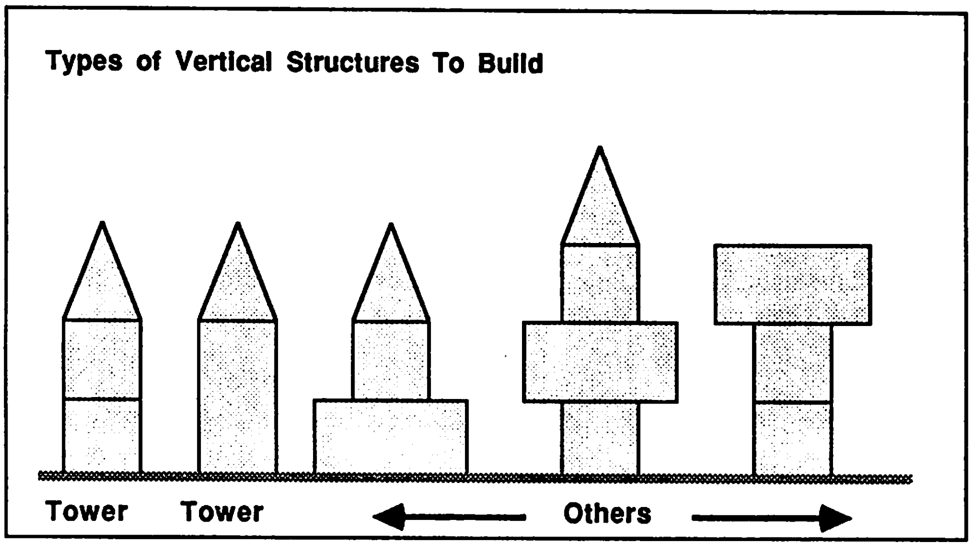


FIGURE 7: Primitive Operators for Extended Blocks World

```
(OPERATOR start-struct IS-PRIMITIVE
; this is an operator for moving a single block on top of another, thereby starting a new
; structure.

(GOAL          top(s,x) AND base(s,y) AND on(x,y))

(PRECOND       (NOT committed(y) , NOT committed (x))
               (STATIC not type-block(y, pyramid)) )

(EFFECTS       (NEW s structure)             (ADD on(x,y))
               (DELETE ontable(x))           (ADD top(s,x))
               (ADD in(s,x))                 (ADD base(s,y) )
               (ADD in(s,y))                 (SET (type-struct s unknown)) )

(OPERATOR extend-struct IS-PRIMITIVE
; this is an operator for moving a single block on top of another, as part of extending an
; existing structure.   Structures cannot be extended if they have a pyramid at the top.

(GOAL          top(s,x) AND on(x,y))

(PRECOND       (NOT committed(x) , top(s,y) )
               (STATIC NOT type-block(y, pyramid)) )

(EFFECTS       (ADD on(x,y))                 (DELETE ontable(x))
               (ADD top(s,x))                (ADD in(s,x))
               (DELETE top(s,y) )            (SET (type-struct s unknown )) ) )

(OPERATOR remove-from-struct IS-PRIMITIVE
; this is the basic operator for taking one block out of a structure.    If there were only two
; blocks in the structure, we disband the structure.

(GOAL          NOT committed(x) )

(PRECOND       (top(s,x))
               (STATIC on(x,y) ) )

(EFFECTS       (DELETE top(s,x))             (ADD clear(y))
               (DELETE in(s,x))              (ADD ontable(x))
               (ADD IF (OLD(NOT base(s,y))) THEN  top(s,y))
               (DELETE IF (OLD(base(s,y)))  THEN  in(s,y))
               (DELETE IF (OLD(base(s,y)))  THEN  base(s,y)
               (SET (type-struct s unknown)) ) )
```

Figure 8: Extended Blocks World Structures



2.3.2 Formalizing the Semantic Database

2.3.2.1 Predicate Calculus Representation If we start with the ER model, then we need to transform entities, relationships, and attributes into predicates and functions of the predicate calculus in order to write the operator clauses as formulas. We make the obvious translation between the ER model and predicate calculus, as follows:

- For each relationship, define a predicate of the same name with arguments of number and type as in the relationship.
- For each attribute with a true/false value, define a (one-place) predicate of the same name; its argument must be an entity of the appropriate entity type.
- For each attribute whose value is other than true/false, define a predicate of the same name which takes as its arguments an entity of the appropriate entity type and a literal representing the attribute value of interest.
- Define functions for certain relationships, taking one or more entities as arguments and returning the entity which completes the relationship. (This must be done only where the result is unique. Two such functions can be defined for 1-1 relationships; one such function can be defined for a many-1 relationship; no functions can be defined for a many-many relationship.)
- Define further functions, if needed, to retrieve attribute values. If entity type E has an attribute named A, then a function named, say, get-A can be defined with

a single argument which is an entity of type E. The result of the function is the value of attribute A for that entity.

- Define arbitrary predicates with n arguments to represent interesting formulas with n variables composed from the foregoing (extensional) predicates and functions, with qualifiers as needed. We call these predicates intensional, because they are not directly recorded in the database; their truth/falsity can be *computed* from the facts recorded in the semantic database.

- Predefined predicates are provided for basic relationships on attribute values (strings, integers, and booleans): equal is overloaded for all attribute value types. The usual substring, greater-than, less-than, etc. can also be used. These predicates are also intensional.

The predicate calculus formulation of the blocks world was given in Figure 5.

2.3.2.2 Database Constraints Included in Figure 5 are a set of SDB constraints. These constraints must hold for any state of the database. (We do not allow the testing of these constraints in the course of an update transaction, during which time the constraints will almost certainly be violated.) The constraints record all assumptions being made about how the domain is being modeled. If a constraint fails to hold for some state of the database, then either there has been an error of interpretation, or there is an error in the effects clause of some operator. Additionally, when choices are being made between alternative interpretations, those interpretations which lead to the violation of SDB constraints can immediately be rejected. For example:

- The "law of gravity" constraint of Figure 5 states that for all blocks, either they rest on the table or they are supported by another block, but not both. If the operator start-struct had the error of omitting the *(DELETE ontable(x))* from the Effects, then such an error would be caught by testing the law of gravity constraint on the state after a start-struct action.

- Suppose our domain included an entity type E with at least two different attributes (A1 and A2), and we had separate plans which set these attributes. Suppose we had a constraint *FORALL x: greater-than(A1(x),A2(x))*. During recognition, we may see an action involving A1 with a particular value VA, but not know for certain which entity was involved. If A2(E1) is greater than VA, then we can rule out E1 right now, thus using the constraint to reduce the number of possible interpretations. But if A2(E1) is not set, E1 is a valid possibility. Suppose we choose to guess E1. Later, we see an action setting A2, and suppose we know for certain that entity E1 is involved. Now suppose that this leads to a state where, for E1, A2 is greater than A1. We must have made a mistake in guessing E1 on the action setting A1.

An alternative use of the database constraints would be to achieve implied database updates. For example, the law of gravity constraint could be interpreted to mean that any time we delete/add *on(x,y)*, we must add/delete *ontable(x)*. This has the advantage of reducing the number of separate effects which must be written in an operator definition (which might reasonably be assumed to reduce the possibility of the writer making errors in writing the operators.) This is a legitimate issue. However there are other (more direct) ways of achieving this goal, which are described in Section 3.3.1 under extensions. Therefore, we retain the interpretation that database constraints are not to be violated, and if violated, indicate an error (of interpretation or operator writing.)

One special use of constraints is to validate the initial state of the semantic database. In the blocks-world of Figure 5, the initial state will contain no structures if and only if there are no stacked blocks. In particular, the correct state description of an initial state where there are four blocks arranged in two stacks will contain two structures. If an initial state description fails to satisfy the SDB constraints, then there is no guarantee that the operators will work correctly.

A final use of constraints involves their applicability to formal reasoning about operator formulas. The constraint which establishes the equivalence of *NOT committed(x)* with *clear(x) AND ontable(x)* could be used to match a precondition of the first form with an operator whose goal has the second form. This application of database constraints is important to maximizing use of the operators in an operator library. It is discussed further in Section 2.6.4.1 on computing which operators can be used to achieve subgoals and preconditions.

2.3.3 Relationship between SDB and Effects Clause

The effects clause of an operator specifies the state transition to be made when the operator is completed. A state transition consists of individual operations which include creation of new objects in the database, addition of new relationships (predicates), deletion of existing relationships, and setting of attribute values. These database changes are denoted by NEW, ADD, DELETE, and SET operations respectively. The collection of individual operations in an effects clause defines a complete transaction on the database. As has been mentioned previously, some of these operations may be conditional.

2.3.3.1 NEW The NEW operation allows the representation of a state change which includes the creation of new objects in the semantic database. A NEW operation takes as its argument a (variable) name for the new object instance. An example of the NEW operation appears as the first effect of start-struct in Figure 7. Note that the type of the object must be specified in the NEW operation.

In very simple domains, the NEW operation is not needed. The textbook blocks world, with a fixed number of block constants and with no explicit representation of the block structures, does not require a facility to create new database objects. However, most complex domains do need such a facility. Some domains are inherently constructive (software development for one), so that the NEW operation is central to being able to model the domain.

It is sometimes necessary to have the NEW operation be conditional on the existence of objects in the database; for example, the operator writer often wants to say "make a new database object only when one meeting such-and-such a description does not already exist." So, as an optional part of the NEW clause, attribute values and/or predicates can be given to serve as the description of the desired object. If this additional information is given, then it is assumed that an actual new object will be created only when no object already exists with exactly these attribute values and for which the predicates are true; if such an object already exists, then it is bound to the variable name. For example:

```
(NEW x type WITH (attr(x,val), rel(x,y))
```

will not result in a new object being created if there is an object x whose attribute *attr* has the value *val* and for which *rel(x,y)* is true. If there is no x satisfying the WITH condition, then a new object x will be created and two implicit database operations will be performed:

```
(SET attr x val)
(ADD rel(x,y))
```

NOTE: The omission of the converse operation, to delete objects in the database, means that the database is not "garbage collected". Such an operation could easily be added to GPF.

2.3.3.2 ADD/DELETE The ADD and DELETE operations take as arguments an (extensional) predicate or a conditional (extensional) predicate. (Limitation to extensional predicates is necessary because those are the ones explicitly recorded in the database -- the intensional predicates are computed from the extensional ones , and thus can be used for querying but not updating the database). Thus, their form is:

```
                    ADD/DELETE <extensional predicate>

      or            ADD/DELETE IF <cond> THEN <extensional predicate>
                              ELSE <extensional predicate>
```

If the condition evaluates to true, then the relationship of the THEN part is added to or deleted from the database; otherwise, the relationship of the ELSE part is added or deleted. (The else clause is optional.)

ADD and DELETE are used in the all the plans of Figure 7.

The use of ADD and DELETE follows the terminology of the earliest planning work, and evokes the "frame problem". We take DELETE p(x,y) to be equivalent to ADD NOT p(x,y). Further, we assume that it is not an error to ADD a predicate which is already true in the database, nor to DELETE a predicate which is already false in the database. The actual implementation of the semantic database can use either the closed or open world assumptions; this implementation issue is not constrained by the plan formalism.

2.3.3.3 SET The SET operation has the form:

```
                    SET <attr spec>

      or            SET IF <cond> THEN <attr spec> ELSE <attr spec>
```

The attribute specification is a triple, consisting of the attribute name, an object (of a type with that attribute), and an attribute value. The database change is to set the given attribute of the given object instance to the given value. A set operation is actually a shorthand notation for two operations: one to delete the existing value of that attribute for that object, and one to add the specified value as that attribute for that object. This notation insures that attributes are single-valued. By providing the special SET syntax for manipulating attribute values, we are also insuring that the attribute be set to a specific value, rather than being constrained to a range of possible values. (Relaxation of this restriction is discussed in the Section 3.3.2.) An example of an unconditional SET operation appears in all the plans in Figure 7.

2.3.3.4 OLD Because the effects clause deals with a state transition involving a start state and an end state (as opposed to dealing with a new state only), there is a need to refer to attribute values and predicates in the old state or to objects identified by relationships in the old state. Otherwise, for example, it is impossible to phrase a database operation which adds a fixed value to the existing value of a numeric attribute. To accommodate this, we provide a distinguished function OLD, which takes as its argument a database predicate or a database function returning an attribute value or an object instance.

The <cond> construct used in the ADD/DELETE and SET database operations can be conditional on the prior state, but not on the final state. (That is because the purpose of the effects clause is to define a computation of a new state from an existing state). Therefore, this construct *must* take the form OLD(<formula>). We require the OLD to be explicit (see the Observe clause discussed in Section 2.6.2.2 for an alternative use of the conditional in Effects.)

The remove-from-struct operator of Figure 7 has some conditional effects using the OLD construct. There can be two different outcomes of the remove-from-struct operator: that the structure simply has one fewer block, leaving a different block at the top, or that the structure is disbanded, and has no blocks in it at all. Thus, the effects clause takes into account whether or not, in the prior state, the block y (which is under block x) is or is not the base of the structure.

2.3.3.5 Semantics The semantics of the effects clause is defined by the following operational model. Before any part of the effects transaction takes place on the database, each instance of OLD is located, its argument is evaluated (in the context of the current database state) and the entire OLD construct is replaced by the value returned (which could be a database object, an attribute value, or a true/false value). Then the NEW operations are performed as follows: first, all WITH specifications are evaluated in the current database state; then all new objects (for which the corresponding WITH failed to evaluate to true) are created . Then, all other operations (including any implied ADDs, DELETEs, or SETs from the NEW operation) are performed in any order according to the condition given for each operation. That completes the transaction.

With this interpretation, some care must be taken in using OLD in a complex operator. The OLD will be evaluated in a state in which all the (final) subgoals are true, not, for example, in the state in which the precondition was true. However this is the right interpretation of OLD for complex plans. For example, it is possible to write correct plans to keep an accurate count of the number of each different type of structure in the world, even when two or more structures of the same type are being built concurrently. Those problems which do arise can be avoided by writing multiple operators with different (static) preconditions, obviating the need for effects conditional on prior states of the database.

2.4 Decomposition Clause

Complex operators are decomposed into subgoals, such that if each subgoal is achieved, then the goal of the complex operator can be achieved (via the addition in the SDB of the effects, if any, of the complex operator). Thus, complex operators are not defined in terms of other operators, but indirectly through states of the database to be achieved by other operators. This makes for a modular operator library -- new operators can be added without having to change existing operators to mention the new operator names.

To make an interesting blocks-world example, let us define a tower to be a stack of blocks three units high where the top block is a pyramid; we exclude the use of horizontal bars in the tower to ensure its columnar shape. One operator for building a tower is a complex operator with a decomposition clause, constructing a tower from two cubes and a pyramid. This operator has two subgoals, one defining the state where the foundation (a two cube stack) is in place and the other defining the state where the pyramid is on top of the foundation. This operator is given in Figure 9 (it does use some operator features which we have not yet discussed.) An alternative operator with the same goal is also given there; the alternative operator builds a tower from a vertical bar (2 units high) and a pyramid (adding the third unit of height). Alt-make-tower has a subgoal decomposition with a single subgoal.

When trying to achieve a complex goal, any of the subgoals which are already true need not be re-achieved. This interpretation of the meaning of the subgoals makes the operators applicable in more circumstances (i.e., larger families of world states). It saves the writer from having to write additional operators which are minor variations of other operators providing for minor variations in the circumstances in which they will be applied.

We treat preconditions as attributes of operators, not of goals or subgoals. So, if there are two operators which achieve the same goal, they need not have the same preconditions. Therefore, no information on subgoal ordering is given in an operator. Subgoals must always be achieved in the order dictated by the preconditions *of the operators which are chosen to achieve them.* On the principle that a plan formalism should not require duplicate information from the writer (thereby avoiding the need both to check for and to resolve inconsistencies), subgoal orderings are not allowed even when all operators for achieving a subgoal have the same preconditions. Orderings are always computed from the relevant preconditions.

FIGURE 9: Operators for Building Towers

```
(OPERATOR make-tower IS-COMPLEX
;we make tower from two cubes and a pyramid.

(GOAL           tower(s) )

(PRECOND        (TRUE))

(DECOMP         (FINAL SUBGOAL build-foundation
                (in(s,x) AND base(s,y) AND on(x,y) )

                (FINAL SUBGOAL add-pyramid
                (in (s,z) AND on(z,x))

(CONSTRAINTS    (type-block(y,cube)) ; base-is-cube
                (type-block(x,cube)) ; middle-is-cube
                (type-block(z,pyramid)) ; pyramid-at-top

(EFFECTS        (SET (type-struct s tower))))


(OPERATOR alt-make-tower IS-COMPLEX
; we make a stack with a vertical bar and a pyramid -- an alternative type
; of tower also 3 units high.

(GOAL           tower(s) )

(PRECOND        (TRUE))

(DECOMP         (SUBGOAL build-it
                (in(s,x) AND base(s,y) AND on(x,y) ))

(CONSTRAINTS    (type-block(y,bar)) ; base-is-bar
                (orient(y,vert)); bar-is-vertical
                (type-block(x,pyramid)) ; pyramid-at-top

(EFFECTS        (SET (type-struct s tower))))
```

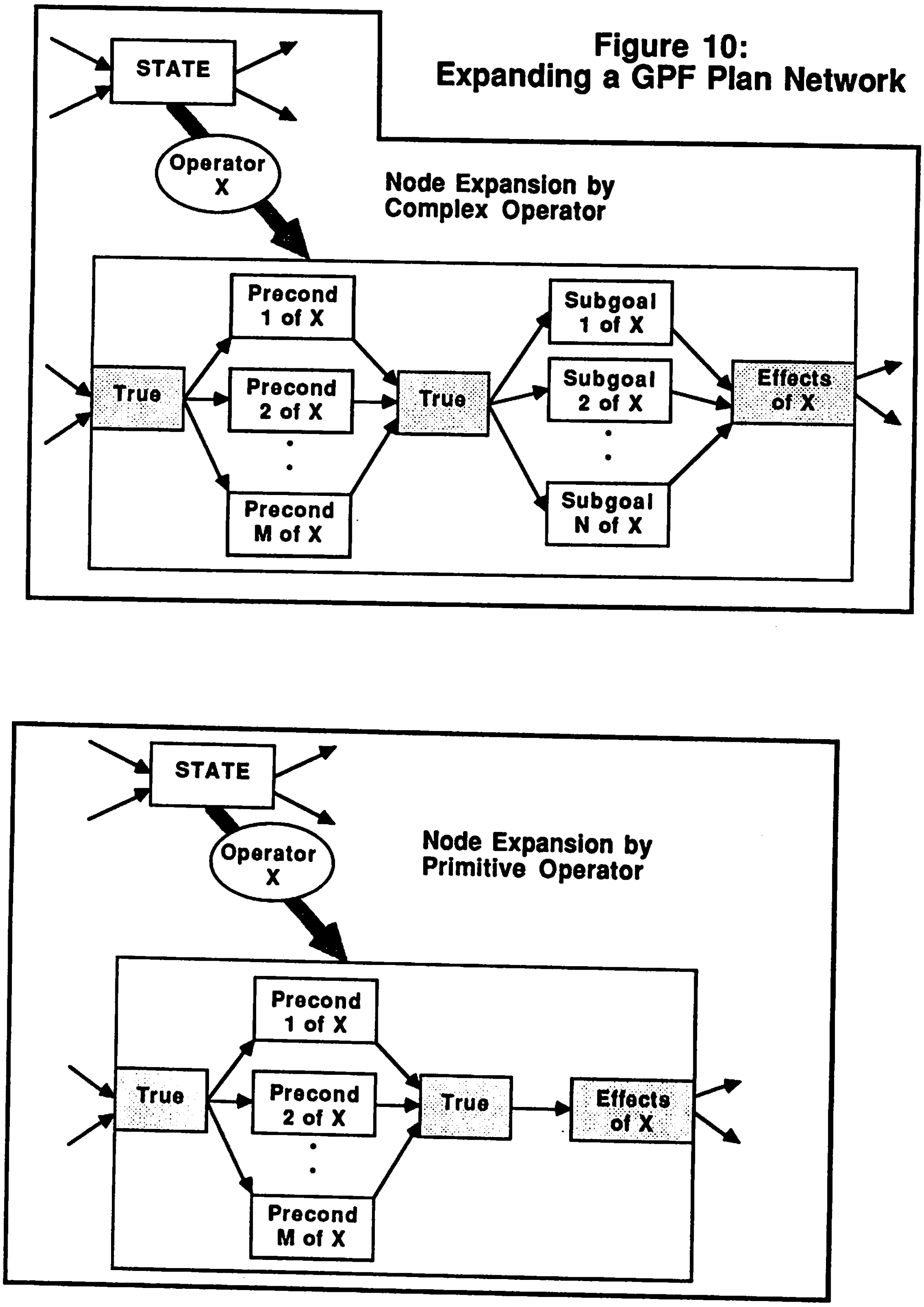
2.4.1 GPF Plan Nets

The standard way to represent hierarchies of plans is through a hierarchical plan network [11]. At each level in a GPF network, there are nodes representing states of the world, tied together in some temporal order (typically a partial order rather than a true linear sequence). Each state is defined by a condition formula: if the condition is true, then the state holds (is achieved.) The states are to be achieved in a sequence dictated by these orderings. If state A has an arrow to state B, then state B must be achieved after state A; when a state has several predecessors, its achievement must take place after all the predecessor states have been achieved. (See also Section 2.6.3 which discusses the durations over which states must be preserved.)

At the highest level in the plan net is a single state representing the goal to be (or being) achieved by this plan net. Top-down expansion from one level to the next is made by selecting an operator to achieve each state appearing at the higher level. The first node of the expansion inherits the predecessors of the higher-level node, and the last node of the expansion inherits the successors of the higher-level node. Certain states, representing effects of operators, are terminal and not subject to further expansion. Other states, whose conditions are already true, also do not need to be expanded (established terminology denotes these as phantom nodes; see also Section 2.6.3 on protection intervals.) Nodes which are not expanded are simply copied from one level down to the next level.

Figure 10 shows the two cases of network node expansion from a higher level to a lower level: via a complex operator and via a primitive operator. In the case of expansion via a complex operator, the expansion includes an operator-head node, followed by multiple nodes in parallel representing the separable parts of the precondition, followed by a precondition-true node, followed by all subgoals in parallel, followed by a operator-end node representing the effects of the operator. (Operator-head and precondition-true nodes act like SPLIT and JOIN nodes [see 11]; their condition formulas are TRUE). In the case of expansion via a primitive operator, the expansion includes an operator-head node, followed by multiple nodes in parallel representing the separable parts of the precondition, followed by a precondition-true node, followed by an operator-end node representing the effects of the operator. In either case, if the precondition is not divided into multiple parts, the nodes preceding the precondition-true node can be omitted and the precondition formula attached to the precondition-true node in lieu of the formula TRUE.

Figure 10:
Expanding a GPF Plan Network



One side-effect of the lack of subgoal ordering in complex operators is that the ordering between the states of the world as seen at any given level in the procedural net may be more permissive than is strictly correct; the orderings that are given are correct, but some orderings may be missing. (These missing orderings are in addition to the missing orderings which result from not yet having considered operator interactions; generating these additional orderings requires further expansion of the plan net.) This permissiveness will be removed at the next lower level in the plan net, when preconditions of the operators chosen to achieve these states are revealed. This is as it should be, since ordering will be dependent upon the operator choices made in the case where multiple operators with different preconditions satisfy the same goal. We have followed a "least-commitment" approach here.

2.4.2 Style of Decomposition

We want to make the choice of subgoals a function of how a typical expert in the domain views the hierarchy of domain tasks. The subgoals should enumerate the significant intermediate database states on the way to achieving the goal. In particular, achieving the precondition to all operators which achieve a subgoal is often an appropriate choice as an additional subgoal. This would usually happen if the achievement of the precondition was not possible via a primitive action, but only via a complex operator. In choosing such a subgoal, the writer is providing additional knowledge to the planning system about aspects which are common to all operators that could be used to achieve another subgoal.

We could have chosen to use additional subgoals in our tower operator, to make the blocks selected for the structure available for the robot to grasp (remember that a block cannot be grasped unless its top is clear). Without additional subgoals, these actions (to clear tops) will still have to be performed, in order to meet the precondition of the operators chosen to build the foundation and add the pyramid. The choices are a matter of style, and a function of how the domain is typically viewed.

We believe, from our own experience writing operators, that writers will often elevate complicated, common preconditions to subgoal level. To make an analogy with an everyday situation, imagine a person writing a list of things to do today. Some entries in the list are likely to be preconditions to other entries, but they are accorded separate entries because they are sufficiently complex in their own right.

In a domain (such as the blocks world example used in this report) where there are a small number of primitive operators, which can be combined in endless ways to achieve higher level goals, the need for including common preconditions as subgoals is not very compelling. However in domains where there are a large number of primitive operators, each used in a distinctive way, this facility will be very useful. This is one case where the blocks world fails to be appropriate to demonstrate GPF features.

In these other types of domains, there will be many cases where a single state expands into a precondition state and a single subgoal satisfiable by a primitive operator; another level in the net is required to reveal the interesting detail in the precondition state. This leads to plan nets which have an excessive number of levels, with each level introducing a very small number of new net nodes. A pathological plan net of this sort, using an example from the software development domain, is given in Figure 11. In this case, allowing common preconditions to appear as subgoals will reduce the number of hierarchical levels, thus reducing the "artificiality" of the hierarchy. Figure 12 shows a case where three levels of the hierarchy are collapsed into a single level.

GPF has been designed so that the operator writer can take advantage of common preconditions when they arise. However, GPF also handles those cases where different operators for achieving the same goal do not have preconditions in common.

2.4.3 Final Subgoals

In order to allow this style of operator writing where preconditions may be elevated to subgoal level, we cannot require that *all* subgoals be true simultaneously in order for the operator to complete. Therefore, we need to distinguish those subgoals which must be true simultaneously in order for the operator to end. We call these the *final* subgoals. The algorithmic identification of which subgoals should be final is non-trivial, due to two factors. First, in the case of an iterated subgoal (discussed in detail in Section 2.4.4), a complex relationship will exist between the subgoal clause and the goal clause. Second, when the goal (and effects) involve domain abstractions, there may be little apparent commonality between the goal and the subgoals (see for example, the tower operator of Figure 9.)

2.4.3.1 Identifying FINAL Subgoals While the algorithmic identification of final subgoals is an open issue, it is straightforward to have the operator writer identify the final

Figure 11:
Pathological Plan Net
Expansion

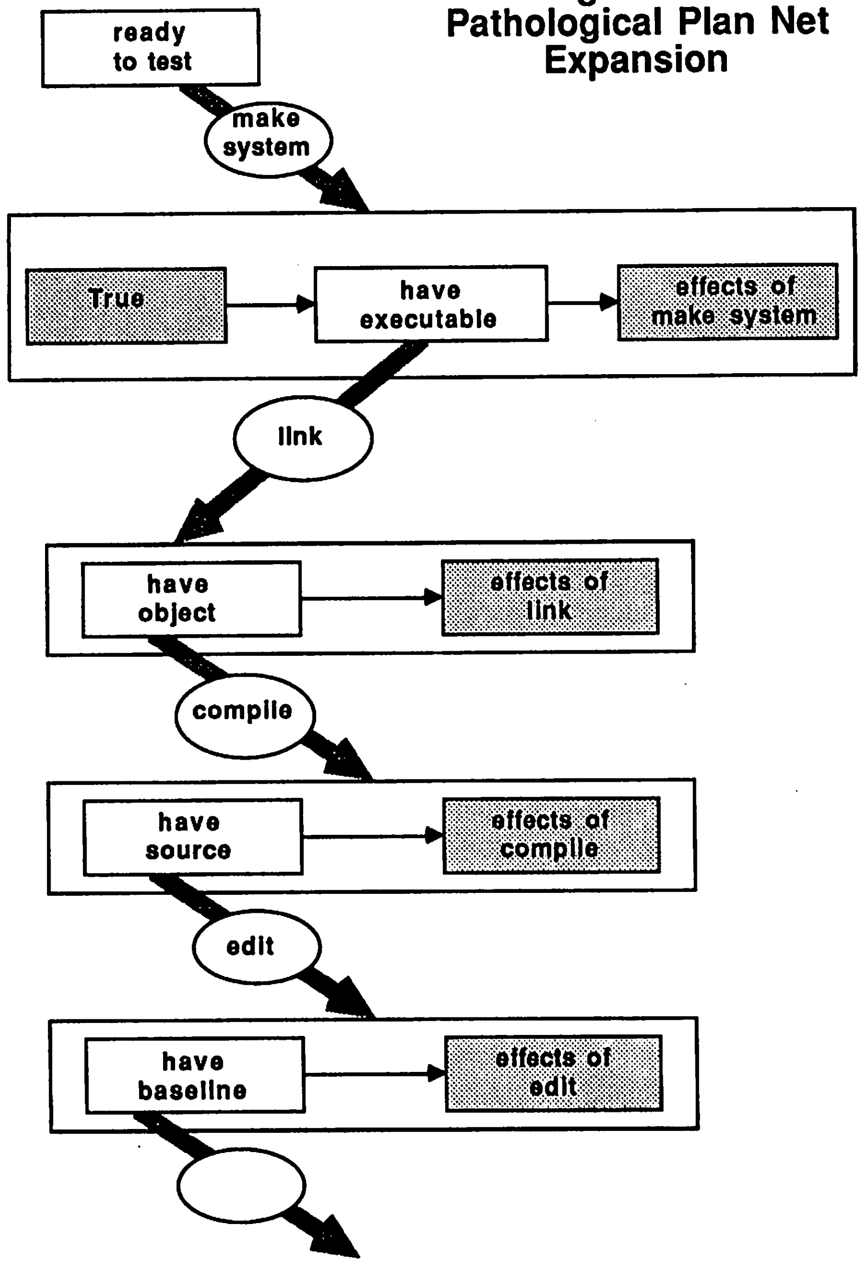
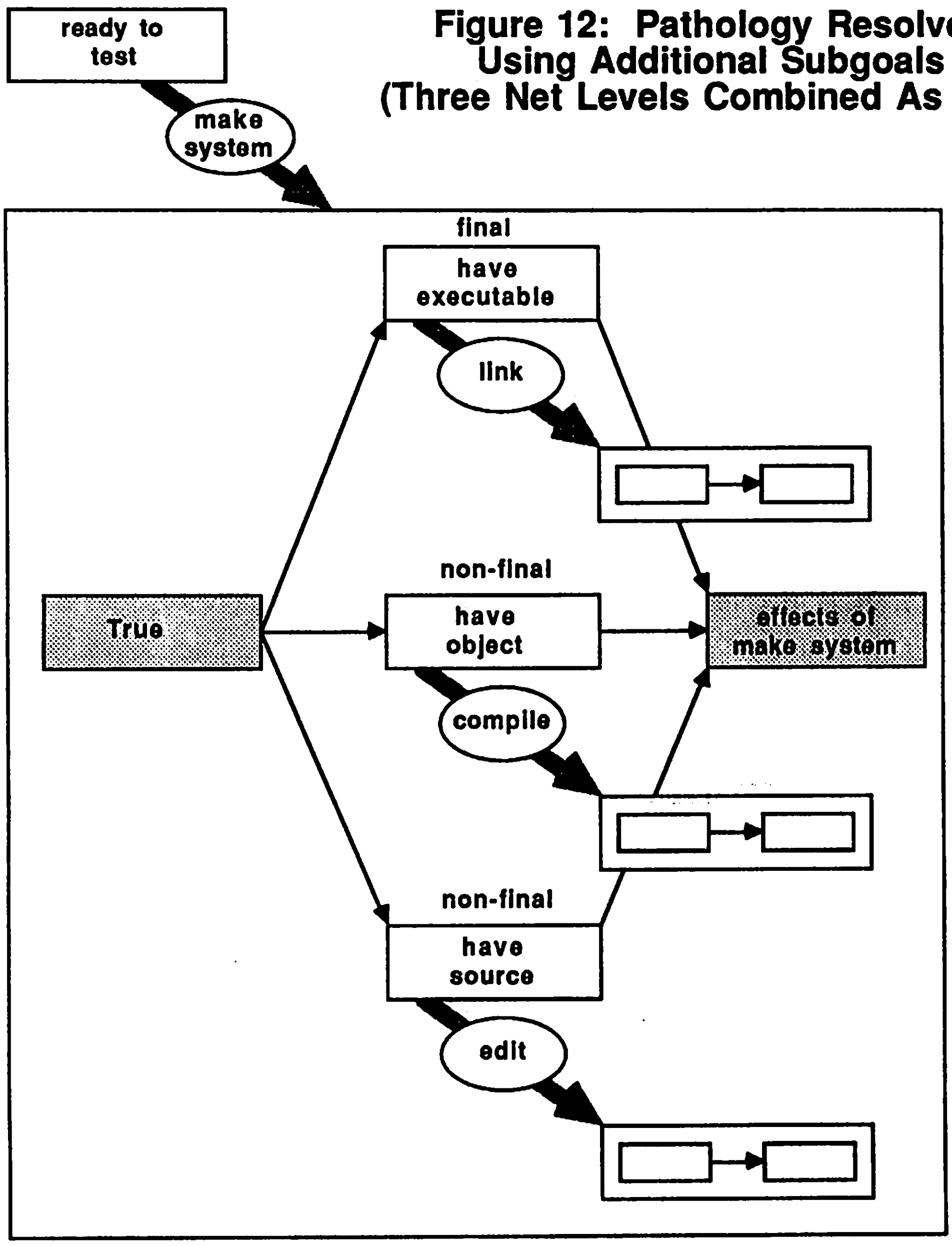


Figure 12: Pathology Resolved Using Additional Subgoals (Three Net Levels Combined As One)



subgoals when the operator is written. A single keyword (FINAL) is all that is required on those subgoals which must be true simultaneously at operator completion in order to ensure that the goal can be made true. The Figure 9 example must have both of its subgoals marked as final.

Subgoals need to be expressed with some care, because (among other reasons) the granularity of FINAL is at the level of the entire subgoal. It would be unacceptable to express the build-foundation subgoal as *top(s,x) AND base(s,y) AND on(x,y)* because *top(s,x)* will become false when the add-pyramid subgoal is achieved. Therefore, we "back off" to the predicate which is not going to change, substituting *in(s,x)* for *top(s,x)* in the complete formula.

2.4.3.2 Examples of Non-FINAL Subgoals As an example of non-final subgoals, consider again the action of constructing a tower. The make-tower operator of Figure 9 can be modified to take advantage of non-FINAL subgoals (the Figure 9 operator is correct as it stands, but not the only way the operator can be written in GPF). There are four additional subgoals needed; three deal with making blocks x,y, and z available to be used in this structure. The other missing subgoal deals with an intermediate step in situations where an existing structure can be modified to construct a tower. (The existing structure might consist of two cubes on which one or more bars had been stacked.) To make this operator follow the decomposition style based on non-FINAL subgoals, we would include additional subgoals as follows:

```
(SUBGOAL make-first-cube-available
NOT committed(x))

(SUBGOAL make-second-cube-available
NOT committed(y))

(SUBGOAL make-pyramid-available
NOT committed(z))

(SUBGOAL remove-extraneous-blocks
top(s,x) )
```

Note that none of these subgoals are FINAL -- they all deal with intermediate steps to be passed through. The remove-extraneous-blocks subgoal requires that in the course of constructing the tower, we pass through a state where block x is on the top of the structure. For a tower made from scratch, this subgoal will be satisfied simultaneously with the build-foundation subgoal, so no additional actions are required. For a tower made by partially dismantling an existing tower (for which build-foundation is already true), this subgoal forces the removal of the extraneous blocks. This subgoal is obviously not a final subgoal, since ultimately z will be placed on x.

2.4.3.3 Other Considerations It should be noted that in some cases, the non-FINAL subgoals will never have to be achieved. In the make-tower example, if build-foundation has already been satisfied, the operator can complete without ever having achieved the subgoals make-first/second-cube-available. An operator always completes when a state is reached in which all FINAL subgoals are true simultaneously, even if some non-final subgoals were never achieved during the course of the operator.

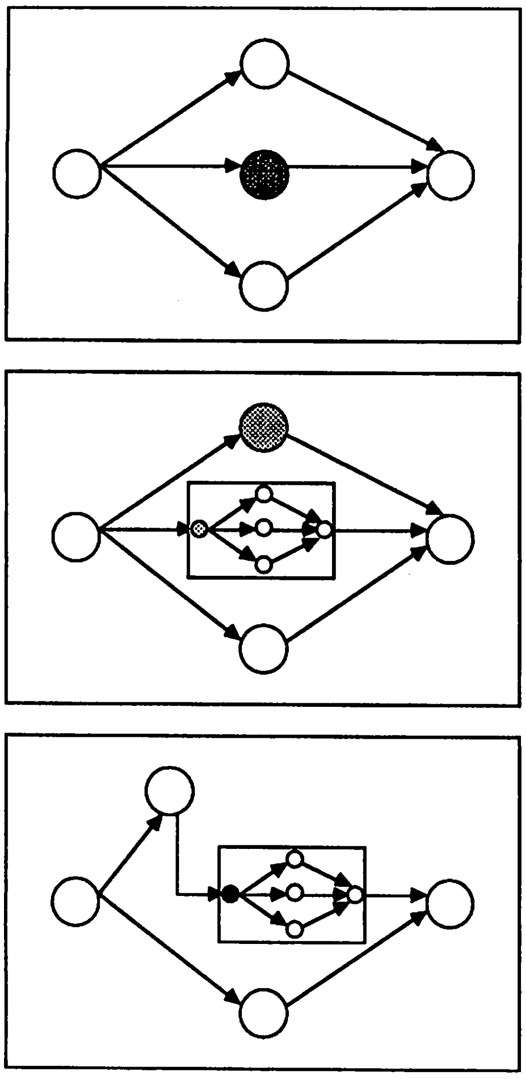
With both FINAL and non-FINAL subgoals all explicit, the operator mirrors the thinking of a person trying to build a tower: "Well, let's see. I've got to make three blocks of the right sort available; that's not simple -- they could be buried deep in existing structures. I've got to stack the two cubes. But, if I can find two cubes already stacked, then I could just remove the extra blocks from that structure. The only other requirement is to place the pyramid. I guess that will do it."

As states representing FINAL subgoals in the plan net are expanded, states will be generated at the lower level which duplicate the non-final subgoal states at the higher level. We obviously want to avoid having duplicate nodes in the net. To accomplish this, the orderings between the nodes at the higher level can be revised so that those (non-final) subgoals which are preconditions to other (final) subgoals are shown to precede them. What would have been the duplicate node becomes a node with a "true" state label (the same as would have been generated for an operator with an empty precondition). This is diagrammed in Figure 13.

2.4.4 Iterated Subgoals

Often, a subgoal must be iterated: the need arises in real world situations to repeat a subgoal with different variable bindings. For example, we could define an operator to partially dismantle a structure, so that a particular block becomes the top block of the (shorter) structure. This operator has a single subgoal, to take a block out of the structure. The subgoal is repeated for each block which is above the block that is to be made the new top block. Another example is an operator to build a tower of arbitrary height; it has one subgoal to start the structure, an

Figure 13: Net Expansions with Non-Final Subgoals


Plan Net to be Expanded
● Node chosen to expand
Node Expanded.
Duplicate Nodes Found:
Non-final Subgoal Same as Newly Revealed Precondition
● Duplicate Nodes
Revision of Plan Net to Handle New Ordering for Non-final Subgoal
● Precondition Changed to True

iterated subgoal to extend the structure, and a third subgoal to place the pyramid on top. In this case the iteration is terminated by the pyramid placement action.

A notation is provided for indicating an iterated subgoal and for specifying its termination. Examples of this notation for the two cases described above appear in Figure 14. In all, there are three types of iteration provided in GPF. The first type is somewhat like a do-until loop. Here, the iteration is terminated when another subgoal becomes true (which may be viewed as being terminated by some condition being achieved, namely that subgoal). This type of

Figure 14: Iterated Subgoal Examples

```
(OPERATOR make-arbitrary-sized-tower IS-COMPLEX

(GOAL              tower(s))

(PRECOND           (TRUE))

(DECOMP            (FINAL SUBGOAL build-foundation
                   (in(s,x) AND base(s,y) AND on(x,y))

                   (FINAL SUBGOAL add-height ITERATED
                   (in(s,z) AND on (z,v)))

                   (FINAL SUBGOAL add-pyramid COMPLETES add-height
                   (in(s,w) and on(w,u))

( .....            ; continues as in previous examples
                   ))


(OPERATOR dismantle-struct IS-COMPLEX

(GOAL              top(s,x))

(PRECOND           (TRUE)
                   (STATIC in(s,x) AND NOT base(s,x)))

(DECOMP            (FINAL SUBGOAL take-off-top
                   ITERATED-OVER (y: above(y,x))
                   NOT in(s,y) ))

; assumes that above(x,y) is intensional predicate as follows:
;     above(x,y) IFF on(x,y) OR (on(x,z) AND above(z,y))
; that is, x is above y if it is directly on y or if it is on a block z which is
; above y.

(EFFECTS           )
```

iteration is used for the arbitrary-height-tower operator. In the second type, iteration is controlled in the manner of a do-loop such as DO FORALL i SUCHTHAT <condition on i>. This type is used for the dismantle-struct operator. A third type of iteration is provided for cases where one subgoal is iterated by being paired with another, so that for each iteration of one, there will be a related iteration of the other. In this case, iteration terminates when the iteration of the paired subgoal terminates (for which a separate termination condition must be specified).

Operators which use the do-until type of iteration are "underspecified". From a plan execution point of view, they do not contain enough information in order to be able to know when to terminate the iteration by executing the completing operator; the user must provide that information. From a plan recognition point of view, the termination of the iteration can obviously be identified when the completing operator is executed. However, there is not enough information to predict when this will happen, nor any information to confirm the "rightness" of termination when it does occur; the user's decision must be accepted without question.

2.5 Constraints Clause

The purpose of the constraints clause is to provide restrictions on valid bindings for operator variables. Such restrictions are necessary to ensure the achievement of the goal. Constraints may involve a single variable, or may define relationships between the two or more operator variables. Constraints are necessary for correctly expressing the operator for making a tower, as given in Figure 9. One successful realization of make-tower is diagrammed in Figure 15. In the tower operator, the constraints serve the purpose of excluding certain unacceptable expansions, as shown in Figure 16. The constraints guarantee that a tower, and only a tower, can be realized from this operator.

2.5.1 Underconstrained Operators

The presence of constraints in an operator definition does not mean that the operator is fully constrained. In the make-tower operator of Figure 9, the choice of blocks from which to make the tower is not specified to the point of uniqueness. Suppose the make-tower operator is executed in a state in which there are exactly two cubes and one pyramid. Two different towers can be built: one with the cubes in reverse order of the other. If there are multiple pyramids in the state, then we have a choice of which pyramid to use for the top of the tower.

Figure 15:
Successful Realization of Make-tower

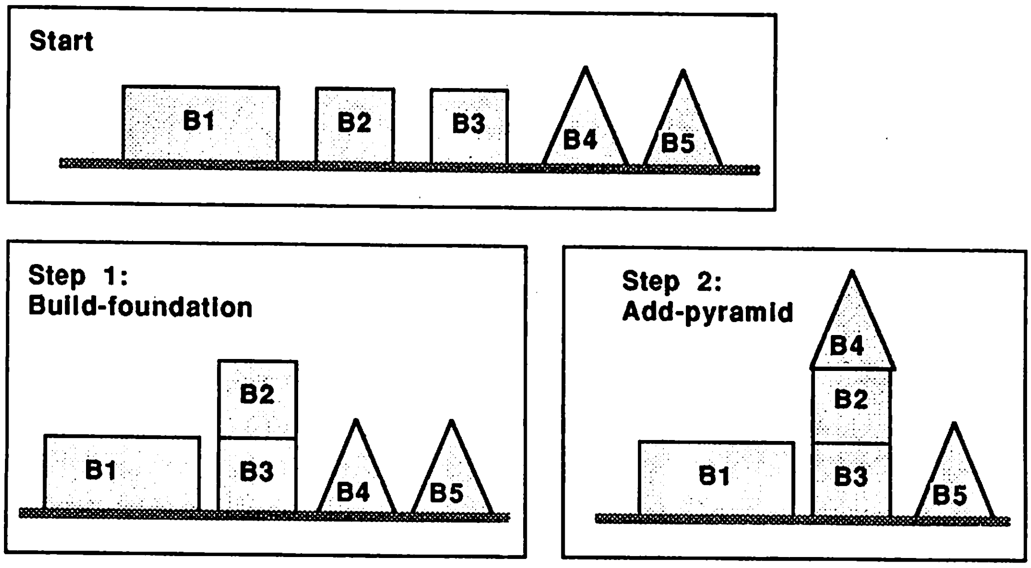
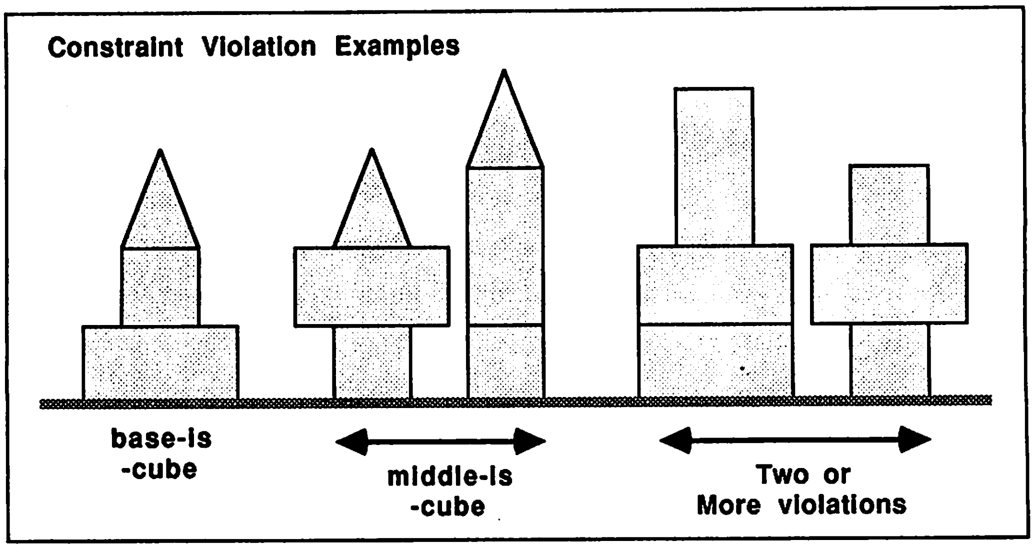


Figure 16: Use of Constraints



An operator which is underconstrained in this way allows the recognizer/planner to make an *arbitrary* choice of binding for the underconstrained variable(s). (Such a choice is always constrained by considerations of other operators in the plan net as well.) But arbitrary choices are not always what is desired. In the types of domains for which GPF has been designed, it is not possible to completely replace the user with an automatic planner. There will be cases where the appropriate constraint cannot be expressed, because the knowledge required to do so is simply not codifiable. In such a case the recognizer/planner must have recourse to the user to make the selection.

In order to distinguish cases where arbitrary choices are not to be made from cases where they can be made, the special constraint OUT-OF-SCOPE is provided. This constraint can be used in addition to other constraints on the same variable. The additional constraints can be checked or used to help predict a binding; but the ultimate binding choice is in the user's hands and cannot be checked for validity. For example, if "OUT-OF-SCOPE(x)" were added to the make-tower operator of Figure 9, then x must be a cube, but only the user has the knowledge to pick which cube to use; the recognizer/planner is prevented from making an arbitrary choice of binding for x.

We call an operator *cooperative* if it uses the OUT-OF-SCOPE constraint on one or more variables. Such an operator cannot be executed automatically without cooperation from the user.

2.5.2 Special Use of Constraints with Iterated Subgoals

In the case of an operator with one or more iterated subgoals, we need to allow some additional notation to be used in the constraints clause. It will usually be necessary to express special constraints for the variables of the first iteration and/or the final iteration; and, there may be constraints which apply between the variables of two successive iterations. Since a single subgoal expression is used to stand for all iterations, it is not possible to express these requirements within the subgoal expression itself. The special keywords FIRST, FINAL, EVERY, THIS, and NEXT along with the predicate "equal" are used for this purpose. "Equal" is a predicate between two operator variables, and is true if both the variables have bindings and those bindings are the same; otherwise, it is false.

Examples of the use of these keywords appear in Figure 17, the full operator for building a tower of arbitrary height. (In the figure, variables have been given descriptive names for improved readability). The constraints relating to the iterated subgoal require that:

- in the first iteration, the newly placed block must rest on the second block which was placed in the foundation step (use of FIRST).
- on the i-th iteration, the newly placed block must rest on the newly placed block of the (i-1)th iteration (use of THIS/NEXT).
- the newly placed block of the final iteration is the block on which the pyramid will be placed (use of FINAL); or, in the case that there are zero iterations to add height, the pyramid is placed on the second block from the foundation step.
- the newly placed block of every iteration must be a cube (use of EVERY).

Figure 17: Constraints in Iterated Subgoals

```
(OPERATOR make-arbitrary-sized-tower IS

(GOAL              tower(s))

(PRECOND           (TRUE))

(DECOMP            (FINAL SUBGOAL build-foundation
                   (in(s,secondblock) AND base(s,baseblock) AND
                   on(secondblock,baseblock))

                   (FINAL SUBGOAL add-height ITERATED
                   (in(s,newblock) AND on (newblock,oldblock)))

                   (FINAL SUBGOAL add-pyramid COMPLETES add-height
                   (in(s,finalblock) and on(finalblock,nexttolast))

 (CONSTRAINTS      (equal(FIRST(oldblock),secondblock))
                   (equal(NEXT(oldblock),THIS(newblock)))
                   (type-block(finalblock,pyramid))
                   (type-block(secondblock,cube))
                   (type-block(baseblock,cube))
                   (type-block(EVERY(newblock), cube) )
                   (equal(nexttolast,LAST(newblock)) OR
                      equal(nexttolast,secondblock) )

 ....              ))
```

2.6 Other Features

2.6.1 On-line versus Off-line Operators

In order to allow a more complete model of domain activities, GPF provides for two distinct types of primitive operators: *on-line* and *off-line.*

2.6.1.1 Definition An on-line operator is one which corresponds to an explicit, monitorable action in the domain. All of the examples given so far in this report have been on-line operators. An off-line operator is an action which is not directly observable, and whose execution must be deduced from evidence of the on-line actions which have occurred.

The intention in introducing off-line operators is to highlight user decisions that play a key role in modeling the domain. For example, consider the issue of designing operators to model the child/robot blocks world. One way to describe the actions of building a tower is "well, *I have to select the blocks that I want to use*, I have to de-commit the selected blocks if they are already in other structures, and I have to build the foundation and add the pyramid." The "action" of selecting the blocks to use corresponds to a user decision which directly affects how the remainder of the operator will be carried out; it does not correspond to any overt and observable action taken by the user.

There is no requirement that off-line operators be used -- highlighting user decisions in this way is a matter of choice relating to the desired view of the domain. The child/robot blocks world as modeled without any off-line operators in Figures 7 and 9 may be a perfectly acceptable approach, depending on the goals of domain modeling.

2.6.1.2 Benefits One advantage of representing user decisions as explicit operators is to have integrated explanations for various (on-line) information gathering activities. In GPF applications, the domain state is so complex that the user typically does not remember it accurately in entirety; this applies both to the domain state as modeled within the intelligent interface and the state information which is outside the scope of that model. Thus, one class of actions commonly performed will be information gathering: explicit probes of the domain state to compensate for the user's imperfect/incomplete memory of that state. For example, weighing a block is performed because the user cannot remember its weight (or perhaps never knew it).

Information gathering actions could be regarded as always legal, meeting high-level goals which are always instantiated. But that interpretation prevents the intelligent interface from correlating information gathering with other ongoing activities. For example, the reason a block is being weighed might be that it is being considered for use in a structure (where say, lighter blocks are preferred to heavier blocks). Given an explicit representation of the decision to select blocks, an association can be made showing the weighing action as part of the decision-making. This is clearly a more interesting model of actions than is possible if information gathering activities are regarded as independent, random, always-legal actions.

Although off-line operators were actually introduced in order to represent user decision making, they can be used for at least one other interesting purpose. Sometimes, actions in a domain may be achievable by either monitorable or non-monitorable means. For example, in computer-based domains, communication between two people could occur by electronic mail, and thus be monitorable. But, it can also occur when the two people meet in the hallway or talk on the phone, and thus not be monitorable. Certain actions (entering a scheduled meeting on one's calendar or fixing a software bug) may be dependent on the fact that communication occurred. To handle this, two different operators are written, each having the same goal (say, learn of meeting or learn of bug); one operator would be on-line and the other off-line. This use of off-line operators extends the action modeling which is possible for the domain.

2.6.1.3 Inferring Execution of Off-line Operators Since off-line operators are not monitorable, their execution must be deduced from the other ongoing activities. This happens when an action is performed whose precondition is not satisfied, but when that precondition could be satisfied by presuming an off-line operator to have occurred. Sometimes the off-line operator will contribute directly to satisfying the precondition, and sometimes the off-line operator will complete another operator which contributes to satisfying the precondition. In the blocks-world case described above, the de-committing of blocks is dependent on (i.e., has a precondition for) the selections having taken place. Thus, when on-line actions to de-commit specific blocks are recognized, it can be inferred that blocks had already been "selected" (and the specific selected blocks can be identified.)

If after-the-fact recognition of off-line operators were the only possibility, then the power gained from defining them would be negligible. However, it will commonly be the case that supporting on-line actions (typically information gathering actions) will be taken *before* the off-line action; by recognizing these related actions, it is possible to *predict* the off-line action (with parameter bindings) before it has occurred rather than to *deduce* it after it has occurred. For example, the decision to use certain blocks in a structure might be dependent upon their distances from one another or their weight (move the lightest, closest blocks in preference to the heaviest, furthest blocks.) Thus, selecting blocks could be represented as a complex operator, with subgoals for measuring and weighing, in addition to a subgoal to make an actual selection. (This last subgoal would be achieved by an off-line operator)

2.6.1.4 Syntax and Operator Writing Style Off-line operators are distinguished in the formalism by a single keyword. Only primitive operators can be denoted off-line. A complex operator may happen to decompose into only off-line operators, in which case it can be thought of as off-line itself. Those complex operators which are potentially off-line (because there is an off-line operator matching each of its subgoals) and those that must be off-line (because only off-line operators match each of its subgoals) can be computed statically.

The operator writer has considerable flexibility in using off-line operators to shape how actions are modeled in the operators. As a matter of GPF operator writing style, we suggest the following possibilities in using off-line operators:

- to represent key user decisions about variable bindings, whether or not those decisions are OUT-OF-SCOPE. In particular, all uses of the OUT-OF-SCOPE constraint can be placed within off-line operators, but off-line operators are not required to contain an OUT-OF-SCOPE constraint. Example: whether or not the intelligent assistant has enough knowledge to bind choices for which blocks to use in a structure, the decision to use a block can be represented by an off-line operator.

- to represent user decisions to terminate a COMPLETES style iteration when no on-line operator can be used to do so. In this case, the operator which signals completion is an off-line operator. Example: The operator to build a tower of arbitrary height has an appropriate completing on-line operator: to place the pyramid. But, an operator to build a column (defined to be a stack consisting only of cubes) of arbitrary height would have an off-line operator to signal that the last cube has been added.

- to create and initialize domain entities which do not have a physical existence in the real world. In some ways this is the converse of the iteration termination: here an off-line operator is used to set up the circumstances in which another operator can be started, rather than completed. This use of an off-line operator helps to bridge the gap between the modeled and unmodeled parts of the domain, or between the physical reality of the world and the user's interpretation of that reality. Example: a separate off-line operator can be written to create structures, rather than having them be created in start-struct.

2.6.1.5 An Example In Appendix B, we present a group of operators for the extended blocks world, including primitive operators and off-line operators which adhere to this style. New structures are created in a separate off-line operator, make-new-struct, instead of in start-struct. Every choice of which block to use is made in the off-line operator, set-block-aside. There are two operators for building a 2-cube/1-pyramid tower: one operator can only be used when there is an existing structure which can be modified into a tower; the other is used when the tower is to be built from scratch.

To write these operators, we introduce a new predicate set-aside, whose arguments are a block object and a structure for which it is "targeted". Basically, we want to separate the notion that a block is set-aside for a structure (the predicate *set-aside*) from the notion of its being available to put into that structure (the predicate *ready*) and from the notion of its actually being part of the structure (the predicate *in*). In the off-line operator, we will set-aside blocks (without placing them); and, in start-struct and extend-struct, we will place blocks (which have already been made ready.)

This example is a bit contrived. It would be more convincing if support activities, such as weighing blocks or measuring distances, had been included. In their absence, there is a lot of extra operator mechanism for rather little gain. We have distinguished certain cognitive actions, but have not provided any extra knowledge about those actions. The example does not contain any cases where an off-line operator is necessary, for example to terminate an iteration. We have made a separate issue of selecting blocks, without giving any information as to the decision-making considerations involved; the set-block-aside operator is as underconstrained as the original formulation.

Off-line operators represent a starting point for deeper domain and user modeling; they are not in themselves the total solution. In some sense, off-line operators are also a place holder for future extensions to the intelligent assistant: we are currently exploring the use of reasoning from first principles to model user decisions.

2.6.2 Interface to Real-world Observations

GPF was designed to be used in an application combining plan recognition, plan execution, and planning. In order to accomplish recognition and execution, an interface to the real world is needed to receive the stream of primitive actions as they occur (recognition) or to generate a stream of the primitive actions to be performed (execution). This interface is not strictly needed for systems which only do planning (i.e., the dynamic construction of plans in the abstract) because a planner typically deals with idealized executions, rather than actual executions including the possibility of failure.

2.6.2.1 Filter Interface We envision a filter program which sends and receives descriptions of primitive actions. During recognition, the filter program is the "window on the world"; during execution, the filter program is the effector of actions as well as a "window" on results. Its purpose is to provide a clean abstraction of real world actions. Within the filter program, alternative action formats and various inconsistencies among formats can be normalized and presented to the intelligent interface in a standardized way. With a suitable abstraction of the real world provided by the filter, the plans themselves become easier to write.

The filter program can be implemented without any knowledge of the semantic database, as it can use strings for object names (which would be represented as attributes of the objects in the semantic database.) We believe that this is an appropriate separation of responsibilities. It implies that all changes to the semantic database are made by/within the GRAPPLE system. Thus, in the operators, we must define database changes for failed as well as successful plan execution; and we must have input from the filter program upon which to base these conditional effects.

An alternative formulation would be to allow the actual semantic database updating to occur independently (perhaps within the filter program), and to define only the effects of success in an operator. In this case, after each operator is executed, the database would be queried to determine if all effects had occurred; if they had, then success is deduced, otherwise failure is deduced.

The advantage of the approach we have chosen is that it is explicit about the nature of the failures which can occur. The disadvantage is that it is very cumbersome (if not impossible) to deal with wide-reaching failures, such as dropping a block (while trying to stack it on another block) in such a way that a tower built earlier is destroyed. This problem derives from the so-called "STRIPS assumption"[7]. This assumption, which is the most commonly chosen approach to the frame problem, provides that all changes between two states be explicitly enumerated in the effects clause and that everything else is assumed not to have changed. Normally it is easy to comply with this assumption in describing state changes for successful plans. But when effects involve non-local changes as in state changes for failed plans, the STRIPS assumption begins to break down. The approach we have taken is perhaps most appropriate when simplified assumptions about the pervasiveness of failure are used.

2.6.2.2 The Observe Clause The Observe clause is the means for realizing the real world interface. This clause is used only in primitive operators; it cannot be used for complex operators, or for any type of off-line operator (for which there is by definition nothing to "observe"). In this clause, the values passed to/from the filter program are specified; each such value is given a name. Since the filter program is to have no knowledge of the semantic database, these values associated with these names are restricted to being of string, boolean, or numeric type; in particular, they cannot be database objects.

For simplicity, we assume that the filter program uses the same operator names as given in the operator definitions. Therefore, no additional information is needed in the Observe clause to name the operation being described.

2.6.2.2.1 Observe Clause Format There are two parts to an Observe clause: that which is used to describe an action (the *stimulus* on the world state), and that which describes its results (the *response* from the world). During recognition, the filter program passes both types of values to the intelligent interface; during execution, the intelligent interface passes the action description values to the filter, which in turn passes the response values back when the action has completed. Response values may be descriptive information which is obtained during an action or they may be success/failure clues. An operator may have no response values, in which case it "cannot" fail; this is appropriate when a somewhat simplified domain is modeled, due to the complexity of the complete domain.

If we model stacking of blocks in the simplest way, then the description of the action (the stimulus) consists of the names of the two blocks in some order (say, base first followed by top block). Thus, the Observe clause would be:

```
        (OBSERVE        (namey namex))
```

and there would be additional constraints as follows:

```
(CONSTRAINTS   (name(y,namey))
               (name(x,namex))
```

The Observe clause establishes that *namex* and *namey* are the variable names by which the values passed to/from the filter program are known; the constraints clause shows how these values relate to other variables already known within the operator.

In this case, there are no response values and failure of stacking is not provided for. We could include a response about block weight (as suggested earlier) as follows:

```
(OBSERVE       (namey namex)
               (RESPONSE measured-weight ))
```

The full version of this operator is given in Figure 18. Here, the variable measured-weight is used for a dual purpose. It is named in the conditional effects, and therefore used for success/failure determination (stacking fails if the measured-weight is over the threshold.) This is an example of a conditional effect which is not conditional upon the prior state of the database, but rather on the outcome of the action as it occurred in the real world. (In this case, we have a conditional effect which does not use the OLD construct.) Measured-weight is also used for the simple feedback of descriptive data, which is recorded in the database in the last Effect.

2.6.2.2.2 User-Supplied Values In some cases, one or more parameters needed to define an action are beyond the scope of knowledge of the planning system, and must be provided by the user. This is analogous to the situation of an out-of-scope constraint on a plan variable, except it applies to a value needed to define an action. In plan execution, this value must be provided before the action can be executed; in plan recognition, the value will be returned as if it were one of the response values. (If only plan recognition were to be supported, no additional construct would be needed.)

A (slightly contrived) example for the child operating the robot can be constructed if we imagine that the robot has both a normal speed and slow motion speed in which any action could be performed; assume one or the other speed must be selected for each operation. Slow motion might be useful when placing a block on a very tall column of stacked blocks which were not precisely aligned on one another. If alignment of blocks is not modeled in the

FIGURE 18: Use of the Observe Clause

```
(OPERATOR Stack IS-PRIMITIVE
; This is variation of the operator of Figure 3.  Here we
; add an observe clause, and use information about the weight of the block.
; Assume that weight is a function of blocks, and its value is one of an
; enumeration consisting of (ok, overweight, unknown)
;(DECLARE             x,y: block)  a sample declaration of operator variables

(GOAL                 on(x,y) )

(PRECOND              (clear(y) , clear(x))
                      (STATIC flattop(y) AND (NOT
                        (equal(weight(x),overweight))) )  )
                      ; notice that we do not use this operator
                      ; if the weight of the
                      ; block is already known to be
                      ; over the threshold

(OBSERVE              ( namey, namex )
                      (RESPONSE measured-weight))
                      ;success of the operator will be dependent on the
                      ; value of measured-weight.

(CONSTRAINTS          (name(y, namey))
                      (name(x,namex)) )

(EFFECTS              (IF (equal(measured-weight,ok) THEN ADD on(x,y))
                      (IF (equal(measured-weight,ok) THEN DELETE clear(y))
                      (IF (equal(measured-weight,ok) THEN DELETE ontable(x))
                      ; if block is too heavy, then the stacked state of the
                      ; blocks is unchanged.
                      (SET (weight x measured-weight)) ))
                      ; in any case, we record the weight
```

semantic database, then the intelligent assistant has no way of deciding when to use slow motion -- only the child can decide.

To handle this situation, an OBSERVE clause construct of the following form is allowed:

```
(OBSERVE          (... USER-SUPPLIED ("description", name) ...)
```

The "description" is needed to support a query to the user for the value during plan execution; the variable name is needed in order to have a name by which to refer to the value (for example, in the effects) during plan recognition. An example use for the case of the two-speed robot described above would be this observe clause for the stack x on y operator:

```
        (OBSERVE        (namex  namey
                        USER-SUPPLIED("standard or slow speed", sp) )
```

2.6.3 Protection Intervals

The partial order information in a plan network specifies valid sequences of achieved states; this is a separate notion from the intervals over which those states must be maintained once they are achieved. For each state in a plan net corresponding either to a normal precondition or to a final subgoal, there is an associated protection interval. Within this protection interval, the state must be maintained: that is, if it is violated, then it must be re-established.

No protection interval is defined for a static precondition: if the static precondition is violated before the operator begins, then the operator choice is no longer valid, and another operator to achieve the same goal must be selected. No protection interval is defined for a non-FINAL subgoal; when a precondition of one of the FINAL subgoals is instantiated which duplicates the non-FINAL subgoal, the two will be coalesced and the protection interval associated with the precondition will be used.

The protection interval for a precondition starts when the precondition becomes true, but not before the associated operator-head node is reached in the partially ordered plan network. This protection interval ends when the operator begins. A primitive operator begins (and also ends) when its effects are posted to the semantic database. A complex operator begins when the first primitive operator in the expansion of one of its subgoals begins.

The protection interval for a FINAL subgoal begins when the subgoal becomes true, but not before the precondition-true node is reached in the partially ordered plan network. This protection interval ends when the operator ends. Any operator (primitive or complex) ends when its effects are posted to the semantic database.

2.6.4 Operator Libraries

An operator library is a collection of primitive and complex operators for a particular domain. An operator library for the extended blocks world, based upon the SDB of Figure 5, is given in Appendix B. This library could easily be extended with the addition of complex operators to build other types of vertical structures than towers; such operators would be modeled on the tower building operators, with different subgoals and different constraints.

Both plan recognition and plan execution call for the construction of (hierarchical) networks of instantiated operators built from the operators in the library. These hierarchical structures can be built in a top-down or bottom-up fashion, or by mixing both strategies.

There is one essential operation in expanding a plan hierarchy: expanding a state which is currently unsatisfied by an operator which will achieve it. Associated with each state is a condition formula defining the state; conditions which are not merely the formula TRUE come in two flavors: preconditions and subgoals. Thus, this matching involves either matching operators to the subgoals they satisfy or matching operators to the preconditions they satisfy. If the expansion is bottom-up, then for a given operator, the idea is to find out which higher level operators it could be part of, i.e., which operators have subgoals that this operator satisfies or which operators have preconditions that this operator satisfies. If the expansion is top-down, then for a given operator, the idea is to find out what lower-level operators are part of it, i.e., which operators could satisfy its precondition and its subgoals.

In computing and using the matches between operators in the library and conditions used in operators in the library, we want to maximize use of the heuristic information supplied by the operator writer about how preconditions can be split into separately achievable parts and about how subgoals represent the significant intermediate step towards achieving the goal. In the remainder of this section, we discuss one set of matching algorithms and its implications on plan writing style. Alternative matching algorithms will be briefly mentioned.

2.6.4.1 Computing Achievers We say that an operator is an *achiever* for a condition when the goal of the operator makes the condition true. Obviously, condition *A* is achieved by an operator whose goal is *A* . But, exact matches are not the only cases of interest. For example, condition *A* is achieved by an operator whose goal is *A AND B*. And, condition *A* is achieved by an operator whose goal is *B* when there is a database constraint of *A IFF B* or *IF B THEN A* .

Achievability is decidable through resolution refutation. Given a condition, we consider each operator goal in turn . If we can derive a contradiction from *NOT (IF <goal> THEN <condition>)* taken together with the semantic database constraints and the definitions of the intensional predicates, then we have found a goal (and thus an operator) which is an achiever for the condition (because the goal logically implies the condition). For example, start-struct is an achiever for the build-foundation subgoal of make-tower: while not identical, the goal of start-struct implies the subgoal build-foundation because *in(s,x)* is implied by *top(s,x)* using one of the semantic database constraints.

Note that with this definition, an operator may fail to satisfy its goal, but still satisfy its purpose (the condition it is an achiever for) in a plan net. For example, an operator P with the goal of *A AND B* could be used as the achiever for a condition *B* ; if P fails because *A* is not achieved, condition *B* is still achieved and P has satisfied its purpose in the plan net. Thus in tracking the progress of operators, the important issue is whether or not the desired condition was achieved, not whether or not the operator succeeded.

When computed as indicated, the set of achievers may include operators which are not appropriate for a given dynamic situation in a plan net. The computation is static, and does not take into account the dynamic picture. In particular, static preconditions and constraints are not considered. For example, take the condition *A(x)* ; if operator A1 achieves *A(x)* with a static precondition of *NOT B(x),* and operator A2 achieves *A(x)* with a static precondition of *B(x),* then we know that A1 and A2 are applicable in mutually exclusive situations. The selection between A1 and A2 must be made dynamically.

It is also important to recognize that the achiever set may be inadequate in a given situation. If the only achiever for *A(x)* has a static precondition of *NOT B(x)*, then there is no operator to apply in the specific situation where *B(x)* is a constraint in the operator in which the subgoal *A(x)* appears. This situation calls for more sophisticated planning.

For a given library of operators, we can define a set of library conditions, made up of all subgoals of complex operators and all (separate parts of) normal preconditions for both complex and primitive operators. For each library condition, the matching algorithm to compute the possible achievers can be executed and the achiever information stored for later use in expanding plan hierarchies. The achievers for all library conditions in the operator library of Appendix B are given in Table 1.

2.6.4.2 Alternative Algorithms The algorithm for computing achievers described here is dependent upon the heuristic information supplied by the operator writer as to how one formula (the entire normal precondition or the goal) can be broken down into separately achievable parts. There is a burden placed upon the operator writer to accurately present the granularity of the separate parts: the plan net can still be correctly expanded if the granularity is too small, but not if it is too big. For example, if a plan P has two subgoals *A* and *B*, and there is a plan Q which achieves *A AND B* as its goal, we will match Q to both subgoals; then, after executing Q to satisfy one subgoal, the other will have been satisfied as well. Thus, when the conditions are broken into more parts than absolutely necessary, the matching algorithm is still effective. Suppose plan P has a subgoal *A AND B* , and we have no match by our algorithm to *A AND B*, but we do have a plan Q for *A* and a plan R for *B*. In this case the algorithm fails to make any connection between Q *together with* R with respect to that subgoal of P. Here the granularity of conditions is too big, and the algorithm fails.

The algorithm we have described takes advantage of the information which GPF makes it possible for the operator writer to provide. However, the use of other algorithms which ignore this information is not precluded. For example, it is possible to use an algorithm which ignores the segmentation of the precondition, treating it as a conjunct of disjuncts to be achieved in parallel (in the manner of STRIPS-style planners [7]).

The algorithm we have described is not completely general, due to several factors. First, it does not take advantage of some dynamic situations. For example, in achieving a condition *A AND B* in the case where *A* is already true, it will ignore operators which achieve *B* alone. Second, it makes no attempt to reason from the effects clause, but always works with the goal clause. This puts some additional burden on the operator writer to phrase the goal accurately; but it also simplifies reasoning -- the effects clause is somewhat awkward to use due to the presence of conditionals.

The advantage of the algorithm described is that it focuses on a small number of possible achievers for any given condition, and thus bounds the search problem in plan expansion. In summary, we consider this algorithm as an appropriate interim approach to expansion of plan

Table 1: Achievers for Subgoals and Preconditions

Plan / Subgoals	**Achievers**
Tower-by-adaptation/	
make-pyramid-available	pick-and-free-block
remove-extraneous-blocks	start-struct*, extend-struct*, dismantle-struct
add-pyramid	start-struct*, extend-struct
Tower-from-scratch/	
get-empty-struct	get-struct
make-first-cube-available	pick-and-free-block
make-second-cube-available	pick-and-free-block
build-foundation	start-struct
make-pyramid-available	pick-and-free-block
add-pyramid	start-struct*, extend-struct
Make-alt-tower/	
make-bar-available	pick-and-free-block
make-pyramid-available	pick-and-free-block
build-it	start-struct
Pick-and-free-block/	
pick	set-block-aside
free	remove-from-struct, make-arbitrary-block-available
Dismantle-struct/	
take-off-top	remove-from-struct, make-arbitrary-block-available
Make-arbitrary-block-available/	
clear-its-top	start-struct*, extend-struct*, dismantle-struct
remove-desired-block	remove-from-struct, make-arbitrary-block-available
Preconditions (normal only)	**Achievers**
start-struct	{pick-and-free-block,pick-and-free-block}
extend-struct	{pick-and-free-block, start-struct}*, {pick-and-free-block, extend-struct}, {pick-and-free-block, dismantle-struct}
remove-from-struct	{start-struct}*, {extend-struct}*, {dismantle-struct}

* Ruled out dynamically when static preconditions and constraints considered

nets: more sophisticated algorithms can be used later. No changes need be made in GPF to use other algorithms.

2.6.4.3 Completeness of Operator Libraries It is instructive for the operator writer to look at the achiever set of any condition (subgoal or element of a precondition), because various omissions/errors can be identified in this way. (Refer to Table 1).

The preconditions of the achievers should cover the appropriate range of cases. For example, the operators make-arbitrary-block-available and remove-from-struct form the achiever set for the goal *NOT committed(x)*. Remove-from-struct is primitive, and requires as a precondition that the block to be removed be at the top of the structure. It would be easy to omit make-arbitrary-block-available, which is a generalization of remove-from-struct that applies to any block in a structure.

The operators which are achievers for a given condition often have some characteristics by which they can be distinguished. For example, start-struct, extend-struct and dismantle-struct are all ways to achieve *top(s,x)*. But start-struct and extend-struct achieve it by adding blocks to a structure whereas dismantle-struct achieves it by removing blocks from a structure. These differences should be reflected in the preconditions of the different operators: in two cases, x must not yet be in s and, in the other case, x must already be in s. Such information is of great value in reducing the number of expansions which are possible from a given node in the plan net.

The achievers should be examined to see if there are missing operators due to goals having been stated too narrowly. The operator writer provides a kind of focusing information when writing the goal of a plan: those details of the effects which are omitted from the goal are implicitly of local, not global, importance. (Remember that the goal focuses on the "main" effects of an operator and separates them from the "side" effects.) However, it is possible for the operator writer to be overly restrictive in stating goals. The result is that an operator which is actually suitable for achieving some condition cannot identified as such because the goal omits the necessary predicate(s). Depending upon the subgoals and preconditions of other operators in the library, the unstack operator of Figure 3b might have an inappropriate goal due to the omission of the predicate *clear(y)* which is one of its effects.

2.7 Use of Predicate Calculus

GPF operators are based upon predicate calculus notation. Each operator clause consists of one or more predicate calculus formulas, with some additional embroidery (GPF keywords like STATIC, NEW, etc, and other constructs such as OLD, conditionals, and iteration specifications, etc.). The formulas are quantifier-free. In this section we discuss how the truth/falsity of these formulas is evaluated using bindings, how sets of bindings called interpretations apply to multiple database states, and how constructing interpretations is related to recognition and execution of plans.

2.7.1 Evaluating Operator Formulas

Bindings bridge the gap between operator formulas and the semantic database, so that the formulas may be evaluated. A binding is a pair <X,Y>, where X is a variable name in an operator and Y is an object in the semantic database. Given a (possibly empty) set of bindings, the truth of the formula *with respect to a particular state of the semantic database* may be determined. A formula is directly evaluatable when all its variables have bindings. When some variables in a formula have no bindings, then existential quantifiers may be added to the formula for all such variables and the truth of the formula may be determined For example, given all the following formulas, bindings and SDB state information:

Plan Formulas	Bindings	Semantic Database
P(x,y)	<x,A1>	P(A1, A2)
Q(z)	<y,A2>	Q(A3)

P(x,y) can be evaluated directly since both x and y have bindings; furthermore, given these bindings, P(x,y) is true. In contrast, Q(z) cannot be evaluated as is, because z is without a binding. However, we can evaluate *THEREEXISTS z / Q(z)*, which will evaluate to true exactly for the binding <z,A3>.

2.7.2 Role of Bindings in Recognition and Execution of Plans

When performing plan recognition, we are given an initial state, and a sequence of actions. From these actions, we want to infer the top-level plan wiiich explains the actions. For example (refer to Figure 19):

- given an initial state of cube C1 on the table, cube C2 on C1, bar B1 on C2, structure ST1 containing C1, C2, and B1, and pyramid P1 on the table
- first action: unstack involving the block named "B1"
- second action: extend-struct placing block "P1" on block "C2"
- we infer the goal to be tower(ST1) via the make-tower plan, where the first action satisfies the remove-extraneous-blocks subgoal, and the second action satisfies the add-pyramid subgoal.

When performing execution of a plan to meet a goal, we are given the goal, an initial state, and need to generate the sequence of actions.

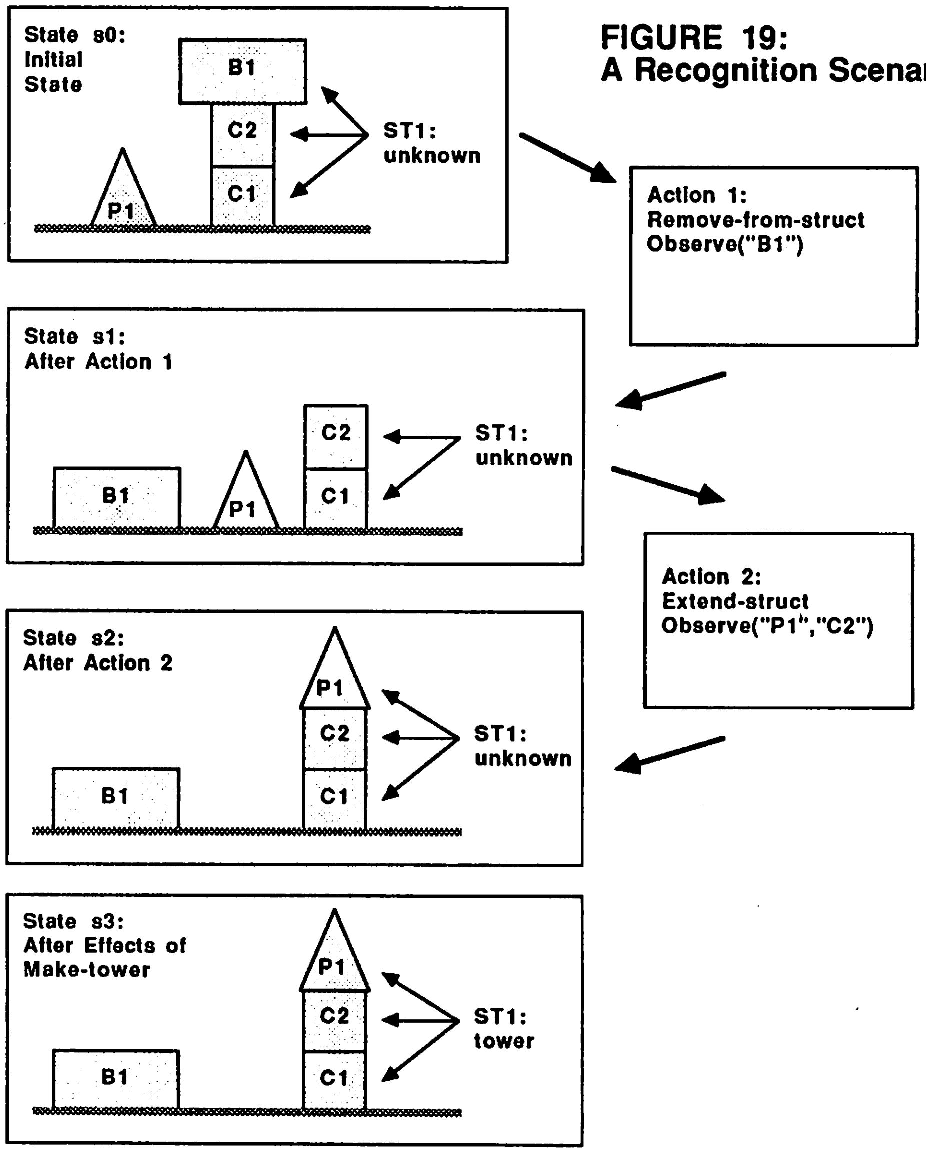


FIGURE 19:
A Recognition Scenario

In performing both plan recognition and plan execution, we are interested in instantiated operators with their variables bound. The operators in the operator library are actually operator templates, representing families of instantiated operators. For example, the unstack operator can be instantiated for any pair of blocks; if the pair is not stacked in some state of the world, then this instantiation of unstack is irrelevant in that state, but may be relevant in some later state.

There are two essential aspects to performing recognition or execution: building a plan net of appropriate operators, and finding the right set of variable bindings for these operators. If the right bindings cannot be found, then other operator choices must be considered. We have already covered (in Section 2.6.4 on operator libraries) which operators need to be considered. In the remainder of this section, we discuss the issue of variable bindings for operators. This issue of bindings is non-trivial in the presence of the constraints clause.

2.7.3 Interpretations and Multiple Database States

Obviously not all the formulas in an operator are intended to be true simultaneously in a single database state. There will be times when both goal and preconditions are false, when the goal is false but the precondition true, when the goal is true and the precondition false, etc. We are interested in sets of bindings for the variables in an operator which have certain properties with respect to multiple database states.

An interpretation consists of a set of bindings for some/all of the variables in an operator. For a given operator, we are (ultimately) interested in an interpretation (call it the goal interpretation) which has a set of bindings for all variables mentioned in the goal statement, and for which the goal statement is true in some (single) database state.

We may be able to arrive at a goal interpretation directly, without executing any operators, when the goal is already true; if not, we use another kind of interpretation, a working interpretation, to try to arrive at a goal interpretation. A working interpretation satisfies the following conditions :

- there is a binding for those variables named in the operator which are needed to make the following true
- there is a database state SP in which the operator precondition is true under this interpretation

- if the operator is complex, then for each final subgoal, there is at least one state SJ in which the subgoal is true under this interpretation. (For iterated subgoals, there are multiple SJ's, one for each iteration necessary to before termination of the iteration).
- if the operator is complex, then for each non-final subgoal, there *may* be a state SK in which the subgoal is true
- if the operator is complex, then there is a state SM in which all final subgoals are true under this interpretation.
- there is a database state SN in which the effects are true under this interpretation.
- the constraints of the operator are true under this interpretation for all database states SP through SN inclusive.
- the states are related in time as follows (see Figure 20):

  * each SJ is at or after SP and at or before SM
  * each SK is at or after SP and at or before SM
  * SM is at or after SP
  * if the operator is complex, SN is after SP and is the state immediately following SM.
  * if the operator is primitive, SN is immediately after SP
- the truth of any operator formula under this interpretation at any time other than that stated above is immaterial.

Figure 20: Time Line

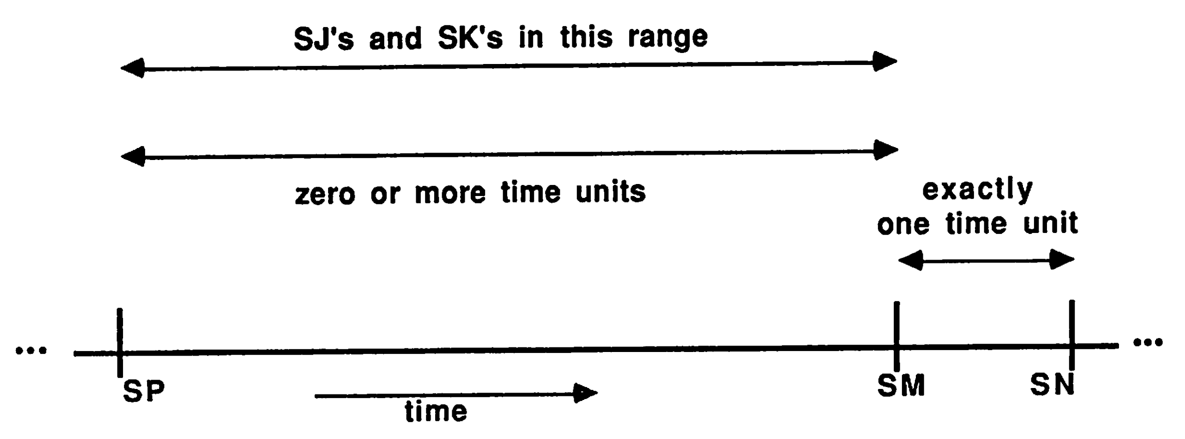
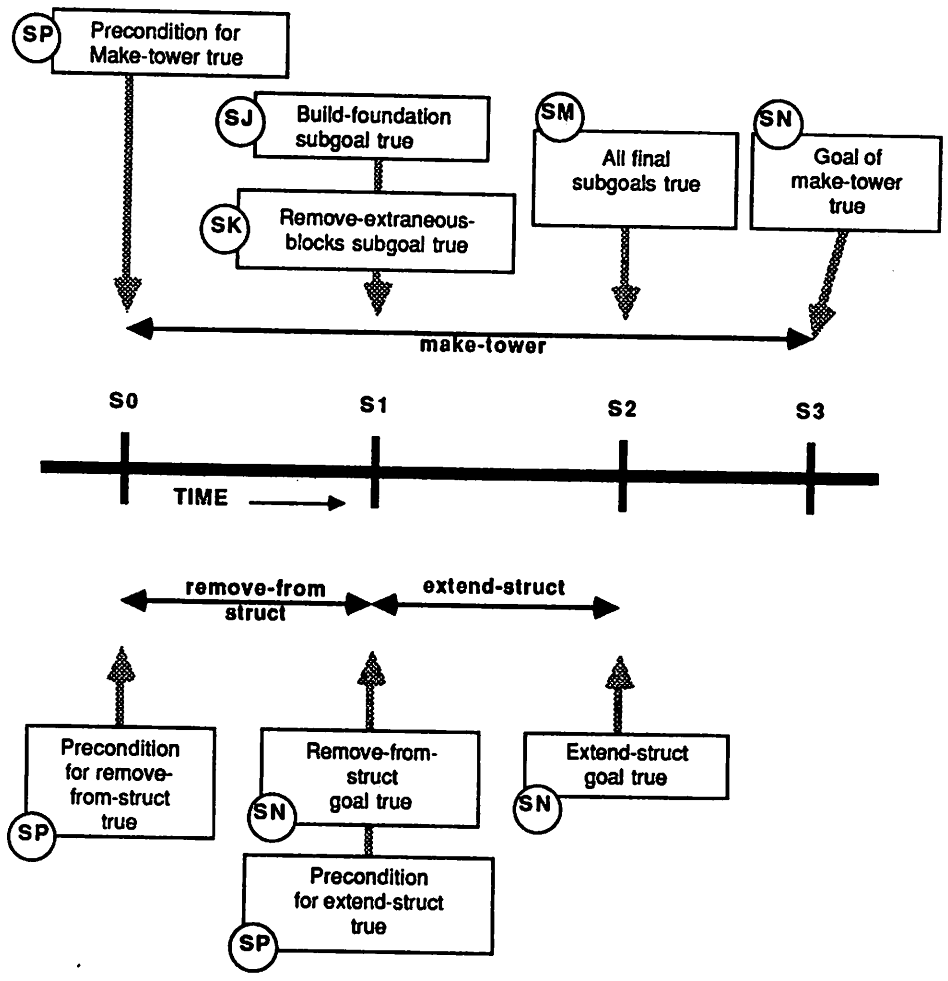


Figure 21: State/Time Diagram


SP
Precondition for Make-tower true
SJ
Build-foundation subgoal true
SK
Remove-extraneous-blocks subgoal true
SM
All final subgoals true
SN
Goal of make-tower true
make-tower
S0
S1
S2
S3
TIME
remove-from struct
extend-struct
Precondition for remove-from-struct true
SP
Remove-from-struct goal true
SN
Extend-struct goal true
SN
Precondition for extend-struct true
SP

If the operator executes successfully, then the goal will be true in state SN and we can derive a goal interpretation from the working interpretation by taking that subset of the bindings which apply to variables in the goal clause.

The working interpretations for the example scenario of Figure 19 are given below (the states are diagrammed in Figure 21):

remove-from-struct	extend-struct	tower
<s, ST1>	<s,ST1>	<s,ST1>
<x,B1>	<x,P1>	<x,C2>
<y,C2>	<y,C2>	<y,C1>
		<z,P1>
SP = state s0	SP = state s1	SP = state s0
SN = state s1	SN = state s2	SM = state s2
		SN = state s3
		SM(build-foundation) = state s0
		SM(add-pyramid) = state s3
		SJ(remove-extraneous-blocks) = state s2
		SJ(make-first-cube-available) = none
		SJ(make-second-cube-available) = none

2.7.4 Constructing Interpretations

We can rephrase the purpose of plan recognitions/execution in terms of goal and working interpretations, as follows. During automatic execution of plans, we are interested in finding a goal interpretation for some operator, given a goal to satisfy and a (possibly empty) set of initial bindings. For the operator library of the extended blocks world, we might be given "tower(s)" as the goal, and optionally some binding for s. If there is no tower in the initial state (i.e., if there is no goal interpretation), then we have the choice of building a working interpretation for make-tower or for alt-make-tower. This in turn means building working interpretations for operators which satisfy the subgoals of these operators. And so forth, down to primitive actions.

During recognition, we are interested in finding a goal interpretation for some top-level operator, given a sequence (not necessarily complete) of primitive actions. Here we want to build a working interpretation of each primitive operator, and working interpretations for the operators they satisfy conditions for, and so on up to a top-level operator.

There are six ways to compute bindings for interpretations: taking them from the initial problem specification, taking them from the filter program, getting values from the user, propagating a binding from another operator by enforcing consistency, performing a NEW database operation in an effect clause, and evaluating operator clauses. We consider each in turn.

2.7.4.1 Values from the Initial Problem Specification In the case of plan execution, some bindings may be supplied as part of the problem definition. For example, we could have been given the problem of generating actions for meeting the goal "tower(s)" given the binding <s,ST1>. Pre-supplied bindings are strictly optional. When they are supplied is determined by what the goal really is. Making the structure ST1 into a tower is a different overall goal from making an arbitrary structure into a tower. We allow all options from fully bound goals to fully unbound goals, and all cases in between.

2.7.4.2 Values from the Filter Program The converse of getting values at the top of the plan hierarchy with the goal (during planning) is receiving values at the bottom of the plan hierarchy from the filter program. This occurs both during plan execution (for RESPONSE-type variables) and plan recognition (for all OBSERVE variables). After an action is executed, its response variables are bound to specific values. When an action is seen, all the observe clause variables are bound by the filter program to specific values.

2.7.4.3 User-supplied Values In the case of an OUT-OF-SCOPE variable, the binding must be supplied by the user. Also, in the case of a USER-SUPPLIED variable during plan execution, the binding must be supplied by the user. In each case, some dialog with the user would take place to establish the binding.

2.7.4.4 Enforcement of Consistency Another source of bindings is through the enforcement of consistency between one operator and another operator which is part of its hierarchical decomposition. The variable bindings must be consistent if the lower level operator is to contribute towards satisfying the higher-level operator. There are two cases for the lower-level operator: either it is a operator to satisfy a precondition, or it is a operator to satisfy a subgoal.

If subgoal A of operator X is being achieved by operator Y, then we say that the interpretations of X and Y are *consistent* if the binding of each variable in the goal of Y is the same as the binding of the corresponding variable in A and the time span S1 to SN of X is within the time span S1 to SM of Y. Similarly, if precondition A of operator X is being achieved by operator Y, then we say that the interpretations of X and Y are consistent if the binding of each variable in the goal of Y is the same as the binding of the corresponding variable in A and the state SN of X equals state S1 of Y. (Correspondence of variables deals with alternate naming schemes: if the goal of Y is *p(x,y)* and the subgoal A of X is *p(a,b)*, then *x* corresponds to *a* and *y* to *b*.). Thus, if we have bindings for X we can propagate them to Y, and vice versa, by using these consistency rules.

2.7.4.5 Performing a NEW Effect By definition, the execution of a NEW operation to create a new database object binds the variable named in the operation to the new object. This means that such an operation will override any binding for that variable which may have been made by other means.

In general it would be desirable to avoid predicting a binding for a variable which will be bound via a NEW effect. However, this is not so easy. For one thing, the NEW can be conditional. For another, there may be several operators for achieving a given goal, and some may have NEW's while others don't.

As an example, consider the make-tower operator (Figure 9) in a world state with one cube on the table and one pyramid on another cube. (Assume the operator library of Figure 7, without off-line plans.) In this initial state, build-foundation is not satisfied, but add-pyramid is. When the build-foundation subgoal is not already true, the only way to make it true is by an operator (start-struct) which unconditionally creates a new structure. Thus, we would like to avoid binding s to the pyramid-cube structure to satisfy add-pyramid, because that binding will have to be retracted when build-foundation is achieved. In practice, there is no way to avoid making the wrong binding for s and retracting it later.

2.7.4.6 Evaluation of Operator Clauses The final way to produce bindings is to evaluate the goal, precondition, constraint, or subgoal clauses of the operator. In general, this technique will provide multiple possibilities for bindings, rather than unique bindings; it is a really a heuristic for predicting bindings. For example, if we are missing a binding for a variable in a precondition, then we can add an existential quantifier for that variable and make a database query of the precondition (as described in Section 2.7.1). All bindings which make the query true are *candidate* bindings for the variable in question.

Candidate bindings are a function of the other bindings used in the query. If those bindings should have to be retracted, then the candidate set will have to be reevaluated. If we arrive at a candidate set *<x,{c1 c2 c3}>* from evaluating *THEREEXISTS x / p(y,z,x)* with *{<y,b1> <z,b2>}*, then if *b2* later proves to be the wrong binding for *z* (because it was itself one of several candidates), the candidate set *{c1 c2 c3}* should be thrown out and the query on predicate *p* reevaluated. In summary, every candidate binding is contingent upon the bindings used in the query which produced the candidate.

There is another contingency assumption involved in candidate bindings: candidates are contingent upon assumptions about the states of the operator to which the clause being evaluated belongs. In the example above evaluating a precondition, there is an assumption that the current state is state S1 of the operator. Such assumptions could later prove to be wrong, in which case the entire candidate set will have to be thrown out and recomputed. (Similarly, evaluation of the goal is contingent upon the assumption that the current state is state SN of the operator; and, evaluation of a subgoal is contingent upon the assumption that the current state is state SM for that subgoal of the operator; and, evaluation of a constraint is contingent upon the assumption that the current state is between states S1 and SN of the operator.)

In summary, the use of evaluation of operator clauses is not guaranteed to return a unique binding. In addition, it is based upon certain implicit assumptions, and thus subject to retraction and reapplication. If candidate bindings are propagated through enforcement of consistency or used to compute other candidate bindings, then the newly computed bindings are also subject to retraction.

The exception to the rule that evaluation of operator clauses provides guesses for bindings occurs when the clause in question involves unchanging aspects of the world state and happens to return a single candidate. The clause could be a static precondition or a constraint. For example, in a blocks world where new blocks are not introduced, and where blocks retain their shape (cube-ness, pyramid-ness, etc.), the selection of the pyramid for the top of the tower is unique if there is but one pyramid in the initial state. It does not matter when we evaluate the clause -- no state change will affect this binding. But, GPF does not provide a general way of determining when this will be the case, so we must treat all bindings resulting from evaluation of operator clauses as if they were tentative.

3.0 Extensions To GPF

The core of GPF, described in the foregoing sections, is not intended to be the final word on a plan formalism for GRAPPLE. It is quite sufficient for a first implementation. In this section, we discuss several types of extensions to GPF which may be implemented in later versions. Most of the extensions add functionality, but a few provide convenience for operator writers without actually making possible the expression of new information about operators.

3.1 Decomposition

We have found that operator writers may find it more natural to make a decomposition of a complex operator via other operators, rather than via subgoals which seem rather abstract. We could allow the following alternative form of a decomposition clause:

```
(Decomp         (OPERATOR <operator name>)
                (OPERATOR <operator name>) ... )
```

The appearance of the keyword OPERATOR (in lieu of the keyword SUBGOAL) would imply that the goal of the named operator is the subgoal, with the meaning that *any* operator which met this goal could satisfy that subgoal. Notice that it is possible to do this only because the externally visible operator variables are exactly those of the goal. Using this approach, the keywords OPERATOR and SUBGOAL could be intermixed within a decomposition. The use of OPERATOR is merely a notational and conceptual convenience to the operator writer.

Having made this step, we can add some additional keywords to restrict selection of operators to meet subgoals. A construct could be introduced, with the form:

```
(ONLY OPERATOR <operator name list>)
```

to indicate that only the named operators could be used to meet this subgoal. The operator name list could have a single operator name or several operator names.

Following the same form, we could allow:

```
(PREFERRED OPERATOR <operator name list>)
```

to mean that the named operators are preferred, but that other operators could be used as well. This construct would provide some focusing information during recognition or execution. However, it is a very simple mechanism; deeper models of operator preferences are definitely more desirable (see Section 3.6).

3.2 Ordering and Forced Execution

The ONLY construct suggested above is a type of escape mechanism for the operator writer, where he can essentially say, "Trust me, this is the way it is. I'm not going to explain it." This is useful when incorporating the explanation into the domain model is deemed to be too complex considering the benefits it brings. Such tradeoffs have to be made when modeling real domains. Two other uses for such an escape mechanism involve additional restrictions of operator ordering and forcing operator execution.

Occasions may arise when the operator writer wishes to impose additional orderings on the sequencing of activities, other than those implied by the preconditions of operators. A simple way of doing this (similar to [14]) is to have a clause in which these additional restrictions can be stated:

```
(SEQUENCE    ((<operator or subgoal name> AFTER <operator
 or subgoal name>) ... ))
```

Partial orders, established pairwise, are very flexible for this purpose. The make-tower plan of Figure 9 could include:

```
(SEQUENCE ( (add-pyramid AFTER build-foundation)))
```

Similarly, occasions arise when an operator writer wants to force the execution of an operator, regardless of whether the goal of the operator is true at the necessary time. We can add the keyword FORCE before the keyword SUBGOAL (or OPERATOR) in a decomposition to achieve this effect.

3.3 Semantic Database Extensions

3.3.1 Macros for Effects Clauses

Writing effects clauses can be tedious. Every time we add *on(x,y)*, we also need to remember to delete *ontable(x)*, or we violate the law of gravity constraint. Such clusters of operations could be expressed as macros so that we could write:

```
(MACRO new-support(a,b) IS
(ADD on(a,b))
(DELETE ontable(a)))

(OPERATOR    start-struct IS-PRIMITIVE
...
(EFFECTS     (INCLUDE new-support(x,y))
             ... )
```

It is true that the use of intensional predicates can also alleviate the proliferation of details in the EFFECTS clause. Intensional predicates are always computed, never explicitly recorded, so making a predicate intensional means it cannot appear in an EFFECTS clause. We could make *ontable(x)* intensional, defined as *FORALL y NOT on(x,y)*. However, if ontable needed to be evaluated frequently, then it is more efficient to record it explicitly as an extensional predicate.

The use of macros is a very direct means of simplifying effects writing. It is not quite so straightforward to achieve this using the constraints alone. In the case of the law of gravity constraint, it not easy to automatically get from the expression of the constraint to something of the form "when the effects include adding/deleting *on(a,b)*, then an implied effect is to delete/add *ontable(a)*". And, if the operator writer should omit the effect dealing with *on(a,b)*, there is no way to bind b from the effect which is provided, namely *ontable(a)*. A constraint of the form *greater-than(a,b)* does not contain enough information to be able to perform any additional database operations to make it true when it is false. Also, not all constraints need be associated with implied database updating; deciding which do and which don't is non-trivial.

3.3.2 Use of Intensional Predicates in ADD/DELETE

GPF operators are precluded from adding and deleting intensional predicates. Thus, we cannot have an effect which includes *ADD committed(x)*. If we were to allow this, then we could have situations where x was committed (because committed(x) is recorded in the database) but *in(s,x)* would be false for all structures s (because we had not also recorded what structure x was committed to). That means that the database is logically inconsistent, given the definition of "committed".

Adding or deleting intensional predicates would be useful in order to have "underconstrained" effects, where part of the detail needed for the complete set of effects is not known. However, to provide this facility, it is necessary to draw upon some sophisticated database/logical techniques. The benefits have to be carefully weighed against the drawbacks (in additional mechanism and additional processing).

A related database extension would be to allow attribute values to be constrained to a range, rather than set to a specific value. This is also useful in situations where information is lacking to make effects fully constrained. This extension requires that predicates on attribute values be able to return a value of "unknown" in addition to "true" and "false". The introduction of "unknown" would have wide-reaching impact on the logical foundation of GPF.

3.4 Specialization Hierarchies

When writing large libraries of operators, it is helpful to be able to define generic operators from which a family of specialized operators can be defined. This saves the operator writer from repeating the generic aspects of the operators in each of the specializations. This facility is similar to the is-a hierarchy which we allow on SDB objects. The simplest provision of such a facility would allow an operator declaration starting :

```
(OPERATOR X IS-SPECIALIZATION-OF Y ...)
```

X would automatically inherit all the clauses of Y. Any clauses given in the definition of X would be added to these base clauses to form the complete operator definition for X. As an added feature, the keyword OVERRIDE could be used to indicate that the corresponding clause of Y is to be replaced entirely by the clause provided in the definition of X.

In Figure 22, we give an example of a specialized operator, make-red-tower, whose definition is based on the make-tower operator. Make-red-tower has an additional constraint: namely, that all the blocks be red; it also has an additional predicate in the goal clause and an additional effect.

Figure 22: Operator Definitions by Specialization

```
(OPERATOR make-red-tower IS-SPECIALIZATION-OF make-tower

(GOAL              AND color(s,red) )

(CONSTRAINTS       (color(x,red))
                   (color(y,red))
                   (color(z,red)) )

(EFFECTS           (SET (color s red))) )

(OPERATOR alt-make-tower IS-SPECIALIZATION-OF start-struct

(GOAL              OVERRIDE tower(s))

(CONSTRAINTS       (type-block(y,bar))
                   (type-block(x,pyramid))
                   (orient(y,vert))

(EFFECTS           (SET (type-struct s tower))))
```

Also, in Figure 22, we rephrase the alt-make-tower operator as a specialization of start-struct, where the base block is constrained to be a vertical bar and the top block is constrained to be a pyramid. In this case, the goal clause is overridden, and there are additional effects.

There is another case where specialization is useful. Strictly speaking, as long as the filter program has no knowledge of the semantic database, it cannot distinguish between a primitive action which is a start-struct and one which is an extend-struct. To deal with this, we can define a single primitive action stack in the manner of Figure 3. Then start-struct can be a specialization; the specialized form will require extra preconditions about x and y both being on the table, and will have extra effects about top, base, and in. Similarly, extend-struct is another specialization, requiring as a precondition that block Y not be on the table, etc.

Operator definitions through specialization could be handled entirely by the operator reader (which takes the external operator form presented here and manufactures the definitions in an internal form), so that the use of specialization is entirely transparent to the recognition/execution algorithms. However, this will prevent us from using specialization to solve the problem of the filter program distinguishing between start-struct and extend-struct (we would have to define these operators as complex ones with a single subgoal satisfiable by the stack primitive operator). In other cases as well, it might be that the existence of specializations might be of some value to the intelligent interface. Expanding the specializations within the operator reader thus is probably not the best way to proceed.

A planning system which is intended to capitalize on specialization hierarchies of operators is described in [15].

3.5 Improved Notation

In GPF, we have chosen to allow each individual operation within an effect clause to be conditional. In practice, operator writers will probably find that the same conditions keep getting repeated. This happens in the stack operator of Figure 18. One solution to this problem is to group multiple operations within the scope of a single condition. This is a fairly straightforward change to the GPF grammar.

There is no way in GPF to assign a name and a value to a variable unless it appears in one of the clauses. For example, we might have an operator with a block variable b such that the effects of the operator included various operations on the top block in the structure to which b belongs. It is cumbersome to have to keep writing *top-of(in-struct(b))* each time we want to refer to this other block. We would like to be able to give it a name, and refer to it by its name each time it is needed. We need a local variable facility to do this in the general case; it could be implemented using macros. In the operator remove-from-struct, the static precondition is expressed as it is only in order to establish a binding for y which can be used in the effects clause; *on(x,y)* logically implies *NOT base(s,x)* which is the more "understandable" expression in this case.

3.6 Specifying Operator Costs

In order to enhance the ability of the intelligent interface to select operators or to predict certain operators in preference to others, information about the costs of different operators is necessary. In some domains, two operators which achieve the same goal may have quite different resource consumption patterns. A generalized clause to do this might look like:

```
        (COST        ( (<resource type> <expression of numeric value>)...)
```

In the blocks world, resources might be time (a function of distance the block is to be moved) and electricity (a function of the weight of the block and the distance moved). In a computer-based domain, resources include CPU time, elapsed wall clock time, user time (to issue command and provide any interactive input), disk space, etc. Domain-specific rules about the relative importance (scarcity) of the different types of resources would also be needed.

3.7 Non-atomic Primitive Actions

In the initial implementations of GPF, we will be treating primitive actions as atomic, that is, as if they occurred instantaneously. Some extensions need to be made to GPF in order to relax this restriction. We will need to know which of the (normal) preconditions for an operator are required simply for the operator to begin, and which must persist until the operator ends. (For complex operators, we assume that a precondition must persist as long as it is needed by an operator achieving one of the subgoals; this is independent of whether or not primitive actions are treated as atomic.)

When primitive actions are not atomic, the stream of actions being recognized or generated will show separate entries for a primitive operator starting and ending. It is interesting to note that there is enough knowledge in GPF as it stands to handle this. At operator start, we need all the non-RESPONSE OBSERVE variables; at operator end, we need only the RESPONSE OBSERVE variables.

3.8 Declarations of Variable Names

In its basic form, GPF has no provision for declaring the types associated with each variable name. Such information is useful for two reasons: it is helpful to the (human) reader of operators and it provides information which can be used for additional checking of the correctness of an operator. If the semantic database is implemented in a strongly typed way (so that a query of the form *on(block,structure)* returns ERROR and not FALSE), then type checking does occur at run-time; with declarations, it could occur at the time the operator library is constructed by the operator reader. If the semantic database is not implemented in a strongly typed way, then no such checking will occur unless there are declarations to be checked at operator-read time.

In the absence of required declarations, the operator writer is always free to include type information in comments. We believe that this is good operator writing style. An example appears in Figure 18.

4.0 Review and Conclusions

In this section we revisit the subject of the plan formalism requirements, and discuss how GPF meets these requirements. We also compare GPF to some of the major milestones in the planning literature.

4.1 How Requirements Were Met

The unique requirement on GPF, to allow for the definition of incomplete operators, is met with three specific GPF features. The three features are the OUT-OF-SCOPE constraint, the COMPLETES form of termination to iterated subgoals, and the USER-SUPPLIED stimulus values defining an action to the filter program; the intelligent assistant is prevented from unilaterally making a binding on variable or value (the first and third cases), or unilaterally terminating an iteration (the second case). One other approach to incomplete plans is possible (in both GPF and other plan formalisms): to define as primitive certain operators which are not actually primitive, thereby requiring the user to supply any necessary substeps and their parameters.

The user-related requirements are met by the use of hierarchical operator definitions, the provision for non-FINAL subgoals (to control the level in the plan hierarchy at which particular conditions appear) and by a style of extended world modeling in the semantic database. (This latter issue is further supported by the use of ER data models to assist with construction of the SDB). Emphasis on the cognitive aspects of the user's actions is provided by the OFFLINE operators.

The real-world requirements are met by operator variables, constraints, and iterated subgoals. The NEW database operation supports domains which are inherently constructive. The RESPONSE variables in an operator provide for the real-world feedback needed both for defining effects of failed operators and for dynamic capture of domain information.

The POISE-related requirements for handling exceptional situations are principally met by introducing goals and preconditions for all operators. Executing an operator is redundant when its goal is already satisfied. Once preconditions are present, the temporal ordering of operators can be deduced. The grammar rules become superfluous under their original interpretation (an ordering to be followed); with their omission, the formalism becomes truly state-based. In any case, the preconditions allow precise control on operator ordering, while the temporal rules approximated the ordering. Concurrency of actions at any hierarchical level is implicit in the fact that any operator whose precondition is true can be executed. The introduction of a goal for each operator allows operator failure to be detected: an operator fails when its goal is not true after its effects are posted. Modularity of operator libraries is ensured because the subgoal decomposition is through states to be achieved by a choice of other operators, not directly through those other operators.

4.2 Relationship to Other Plan Formalisms

The foundation of GPF lies in the basic state-based plan formalisms introduced with the early planning work, such as STRIPS [7]. At the core of such operator definitions are the two clauses defining the preconditions and effects. The so-called STRIPS assumption, requiring enumeration of all database changes comprising a state change, is used in GPF.

The form of GPF operators is intended to capture as much domain knowledge from the operator writer as possible. An important part of this domain knowledge is the subgoal decomposition for complex operators. The subgoals enumerate the intermediate steps which close the distance between an arbitrary present state and the desired final state where the goal of the operator is true. This knowledge bears a resemblance to GPS means/end analysis [10]. A GPS planner identifies differences between the present state and the desired state, and uses these differences to select which operators to apply; the selection is made from a pre-supplied table correlating differences to operators. In GPF the differences are the subgoal states, and the matching operators are the achievers for those states.

The subgoal decomposition of GPF also has similarities to NOAH [11], the key work introducing hierarchical operators and hierarchical networks of plans. NOAH operators are not written from the user's perspective, but from the system's perspective: thus, they are operators for expanding nodes in a net rather than operators for actions in the domain. NOAH makes default assumptions about the intervals over which conditions must be preserved; these default assumptions are not general. These two characteristics of NOAH operators were improved upon in the NONLIN [14] system. NONLIN, and to some extent NOAH, require the operator writer to give ordering information which in GPF is deduced from normal preconditions and the fact that a normal precondition is a distinguished subgoal which must precede all other subgoals.

SIPE [16] is a recent and very full-featured plan formalism built on the same base of earlier work as GPF. GPF constraints are similar to SIPE's (and both owe something to MOLGEN [12,13]), but SIPE has some additional constraint types not in GPF. SIPE provides several of the extensions to GPF defined in Section 3, including the decomposition through operators. SIPE has an elegant approach to simplifying effects clauses through "deductive operators" (which are also used to achieve conditional effects). SIPE has a special language for operator resources, and special algorithms to support reasoning about resources. It also provides a way to distinguish changeable and unchangeable aspects of the world state; these aspects are implemented differently to improve efficiency.

Three aspects of GPF are new: the ability to specify incomplete operators, the notion of offline operators and the provision for database changes which create new database objects. The treatment of iterated subgoals in GPF is more extensive than that of other formalisms. The emphasis on user-oriented domain modeling, supported by the use of the ER model of data, is unique to GPF. One difference of style from NOAH, NONLIN, and SIPE is that GPF does not require the operator writer to specify ordering or purpose interrelationships between subgoals and preconditions.

4.3 Acknowledgments

All of the other members of the GRAPPLE project have contributed to the development of GPF. Carol Broverman raised many insightful questions during numerous design discussions. She and Chris Eliot have taken the lead in designing the plan recognition algorithms for GPF. Bob Cook completed the initial implementation of the recognition algorithms. Prof. Bruce Croft provided comments on an initial draft of this report.

5.0 References

[1] Broverman, C.A., and W.B. Croft, "A Knowledge-based Approach to Data Management for Intelligent User Interfaces", *Proceedings of Conference for Very Large Databases*, 1985.

[2] Broverman, C.A., K.E. Huff, and V.R. Lesser, "The Role of Plan Recognition in Intelligent Interface Design, *Proceedings of Conference on Systems, Man and Cybernetics*, IEEE, 1986, pp. 863-868.

[3] Carver, N., V.R. Lesser and D. McCue, "Focusing in Plan Recognition", *Proceedings of AAAI*, 1984, pp. 42-48.

[4] Chen, P.P., "The Entity-relationship Model: Toward A Unified View of Data," *ACM Transactions on Database Systems*, vol. 1, no. 1, March 1976, pp. 9-36.

[5] Croft, W.B., L.S. Lefkowitz, V.R. Lesser and K.E. Huff, "POISE: An Intelligent Interface for Profession-Based Systems", *Conference on Artificial Intelligence*, Oakland, Michigan, 1983.

[6] Croft, W.B., and L.S. Lefkowitz, "Task Support in an Office System", *ACM Transactions on Office Information Systems*, vol. 2, 1984, pp. 197-212.

[7] Fikes, R.E., and N.J. Nilsson, "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving", *Artificial Intelligence*, vol. 2, 1971, pp. 189-208.

[8] Huff, K.E. and V.R. Lesser, "Knowledge-based Command Understanding: An Example for the Software Development Environment", Technical Report 82-6, Department of Computer and Information Sciences, University of Massachusetts, Amherst, 1982.

[9] Huff, K.E. and V.R. Lesser, "Intelligent Assistance for Programmers Based Upon a Formal Representation of the Process of Programming", Technical Report 86-09, Department of Computer and Information Sciences, University of Massachusetts, Amherst, 1986.

[10] Newell, A. and H.A. Simon, "GPS: a Program that Simulates Human Thought", in Feigenbaum, E. and J. Feldman eds., *Computers and Thought*, McGraw-Hill, New York, 1963.

[11] Sacerdoti, E.D., *A Structure for Plans and Behavior*, Elsevier-North Holland, New York, 1977.

[12] Stefik, M., "Planning with Constraints", *Artificial Intelligence*, vol. 16, 1981, pp. 111-140.

[13] Stefik, M., "Planning and Meta-planning", *Artificial Intelligence*, vol. 16, 1981, pp. 141-169.

[14] Tate, A., "Project Planning Using a Hierarchical Non-linear Planner", Dept. of Artificial Intelligence Report 25, Edinburgh University, 1976.

[15] Tenenberg, J. "Planning with Abstraction", *Proceedings of AAAI*, 1986, pp. 76-80.

[16] Wilkins, D.E., "Domain-Independent Planning: Representation and Plan Generation", *Artificial Intelligence*, vol. 22, 1984, pp. 269-301.

Appendix A
Formal Grammar for Operator Definitions

Grammatical Notes

Symbols:	[...] denotes an optional construct
	l denotes alternation
	{ } used for bracketing
Comments:	Comments within an operator definition will follow the Common LISP convention: from a semi-colon to the end of the line.

Grammar

```
<operator>            <-    ( OPERATOR <operator name>
                            {IS_COMPLEX l IS_PRIMITIVE} [OFFLINE]
                            <goal clause>
                            <preconds clause>
                            [<decomp clause> l <observe clause> ]
                            <constraints clause>
                            <effects clause> )

<goal clause>         <-    ( GOAL <formula>)

<preconds clause>     <-    ( PRECOND (<formula list>) [(STATIC <formula>)] )

<formula list>        <-    <formula>, <formula list> l <formula>

<decomp clause>       <-    ( DECOMP <subgoal deflist> )

<subgoal deflist>     <-     <subgoal def >
                            l <subgoal def> <subgoal deflist>

<subgoal  def>        <-    ( [FINAL] SUBGOAL < subgoal name >
                            [<iteration spec>]
                            < formula > )

<iteration spec>      <-    ITERATED
                            l COMPLETES <subgoal name>
                            l ITERATED-OVER (<vbl name> : <set spec>)
                            l PAIRED-WITH <subgoal name>

<set spec>            <-    <formula>

<observe clause>      <-    (OBSERVE (<ob name list>) [( RESPONSE <name
                            list>)])
```

```
<ob name list>           <-    <vbl name>
                               | <vbl name> <ob name list>
                               | <user supplied value>  <ob name list>

<user supplied value>    <-    USER-SUPPLIED ( <descript>, <vbl name> )

<descript>               <-    <string>

<name list>              <-    <vbl name> | <vbl name> <name list>

<constraints clause>     <-    ( CONSTRAINTS <constraint list> )
                               | (CONSTRAINTS)

<constraint list>        <-    < constraint>
                               | <constraint> <constraint list>

<constraint>             <-    <formula>

<effects clause>         <-    (EFFECTS <effects list>)
                               | (EFFECTS )

<effects list>           <-    <effect>
                               | <effect> <effects list>

<effect>                 <-    ( <newobj op>
                               | <add-delete op>
                               | <attr set op> )

<add-delete op>          <-    ADD | DELETE <cond-predicate>

<cond-predicate>         <-    <predicate>
                               | IF <formula> THEN <predicate>
                               |  IF <formula> THEN <predicate>  ELSE
                                  <predicate>

<newobj op>              <-    NEW < obj name> <type name> [WITH (<with spec>) ]

<with spec>              <-    <predicate> | <predicate>,<with spec>

<attr set op>            <-    SET <attr spec>
                               | SET IF <cond formula> THEN < attr spec >
                               | SET IF <cond formula> THEN < attr spec >
                                   ELSE < attr spec >

<attr spec>              <-    (<attr name> <vbl name> <value>)

<formula>                <-    (<formula>)
                               | OLD ( <formula> )
                               | <predicate> | NOT <predicate>
                               | <formula> AND <formula>
                               | <formula> OR <formula>
```

```
<predicate>        <-     <pred name> ( <arg list>)
                          | TRUE | FALSE

<arg list>         <-     <arg> | <arg> , <arg list>

<arg>              <-     <predicate>
                          | <vbl name> | <qualifier> (<vbl name>)
                          |<funct>
                          | <value>

<qualifier>        <-     THIS | NEXT | FIRST | FINAL | EVERY | SOME

<funct>            <-     <funct name> ( <arg list> )
```

Appendix B
An Operator Library

The following additional SDB predicates and constraints are assumed, in addition to those of Figure 5:

Extensional:	setaside(structure,block)
Intensional:	ready(structure,block): ready(s,y) IFF setaside(s,y) AND NOT committed(y)
	free(block): free(y) IFF FORALL s: NOT setaside(s,y)
	empty(structure): empty(s) IFF FORALL b: NOT in(s,b)
Constraints:	setaside(s1,b) AND setaside(s2,b) IFF equal(s1,s2)

```
oooooooooooooooooooooooo

(OPERATOR start-struct IS-PRIMITIVE
; this is an operator for moving a single block on top of another, to be used
; when the structure being built is currently empty of blocks.

(GOAL              on(x,y) AND top(s,x) AND base(s,y) )

(PRECOND           (ready(s,y) , ready(s,x))
                   (STATIC NOT type-block(y, pyramid)
                   AND empty(s)) )

(OBSERVE           (namex,namey) )

(CONSTRAINTS       (name(x,namex))
                   (name(y,namey)) )

(EFFECTS           (ADD on(x,y))
                   (DELETE ontable(x))
                   (ADD top(s,x))
                   (ADD in(s,x))
                   (DELETE clear(y))
                   (ADD base(s,y) )
                   (ADD in(s,y))
                   (SET (type-struct s unknown))
                   (DELETE setaside(s,x))
                   (DELETE setaside(s,y))) )
```

```
(OPERATOR extend-struct IS-PRIMITIVE
; this is an operator for moving a single block on top of another, to be used
; when a structure already has blocks in it.

(GOAL           on(x,y) AND top(s,x) )

(PRECOND        (top(s,y) , ready(s,x))
                (STATIC NOT type-block(y, pyramid)
                AND NOT empty(s) AND NOT in(s,x)) )

(OBSERVE        (namex,namey) )

(CONSTRAINTS    (name(x,namex))
                (name(y,namey)) )

(EFFECTS        (ADD on(x,y))
                (DELETE ontable(x))
                (ADD top(s,x))
                (ADD in(s,x))
                (DELETE top(s,y))
                (SET (type-struct s unknown))
                (DELETE setaside(s,x)) )
```

```
(OPERATOR remove-from-struct IS-PRIMITIVE
; this is the basic operator for unstacking, i.e., taking one block out of a
; structure.   If there were only two blocks in the structure, we disband
; the structure.

(GOAL           NOT committed(x) AND NOT in(s,x) )

(PRECOND        (top(s,x))
                (STATIC on(x,y) AND in(s,x)) )

(OBSERVE        (namex) )

(CONSTRAINTS    (name(x,namex)) )

(EFFECTS        (DELETE top(s,x))
                (DELETE in(s,x))
                (ADD IF NOT(OLD(base(s,y))) THEN
                 top(s,y))
                (DELETE IF (OLD(base(s,y))) THEN
                 in(s,y))
                (DELETE IF OLD(base(s,y)) THEN
                  base(s,y)
                (ADD clear(y))
                (ADD ontable(x))
                (SET (type-struct s unknown))) )
```

```
(OPERATOR tower-by-adaptation IS-COMPLEX
;we make a tower from an existing structure which has a cube at its base
; and a cube on top of the base.

(GOAL            tower(s) )

(PRECOND         (TRUE)
                 (STATIC base(s,y) AND on(x,y)))

(DECOMP          (SUBGOAL make-pyramid-available
                 ready(s,z) )

                 (SUBGOAL remove-extraneous-blocks
                 top(s,x))

                 (FINAL SUBGOAL add-pyramid
                 (top (s,z) AND on(z,x))

(CONSTRAINTS     (type-block(z,pyramid)) ; pyramid at top
                 (type-block(x,cube)); cube in middle
                 (type-block(y,cube)); cube at base

(EFFECTS         (SET (type-struct s tower))) )
```

```
(OPERATOR tower-from-scratch IS-COMPLEX
;we make a tower from scratch.

(GOAL           tower(s) )

(PRECOND        (TRUE))

(DECOMP         (SUBGOAL get-empty-struct
                empty(s))

                (SUBGOAL make-first-cube-available
                ready(s,x))

                (SUBGOAL make-second-cube-available
                ready(s,y) )

                (FINAL SUBGOAL build-foundation
                (on(x,y) AND in(s,x) AND base(s,y) )

                (SUBGOAL make-pyramid-available
                ready(s,z))

                (FINAL SUBGOAL add-pyramid
                (top (s,z) AND on(z,x))

(CONSTRAINTS    (type-block(z,pyramid)) ; pyramid at top
                (type-block(x,cube)); cube in middle
                (type-block(y,cube)); cube at base

(EFFECTS        (SET (type-struct s tower))) )
```

```
(OPERATOR make-alt-tower IS-COMPLEX
; we make a stack with a vertical bar and a pyramid -- an alternative type
; of tower also 3 units high.

(GOAL            tower(s) )

(PRECOND         (TRUE))

(DECOMP          (SUBGOAL make-bar-available
                 ready(s,y) )

                 (SUBGOAL make-pyramid-available
                 ready(s,x))

                 (FINAL SUBGOAL build-it
                 (top(s,x) AND base(s,y) AND on(x,y) ))

(CONSTRAINTS     (type-block(y,bar)) ; base-is-bar
                 (orient(y,vert)); bar-is-vertical
                 (type-block(x,pyramid)) ; pyramid-at-top

(EFFECTS         (SET (type-struct s tower))) )
```

```
(OPERATOR make-new-struct IS-PRIMITIVE OFFLINE
;this is an operator to find an empty structure, i.e. to get a "label"
;which can be attached to a group of blocks that will be stacked.
;if there are no empty structures, then a new one is created

(GOAL            (empty(s))

(PRECOND         (TRUE))

(CONSTRAINTS     )

(EFFECTS         (NEW s structure))
```

```
(OPERATOR set-block-aside IS-PRIMITIVE OFFLINE
;this is an operator to target a block for a structure being built.
;both uncommitted blocks and committed blocks are candidates for
;being set aside; but blocks already set aside are not.

(GOAL           (setaside(s,x))

(PRECOND        (TRUE)
                (STATIC free(x)))

(CONSTRAINTS    )

(EFFECTS        (setaside(s,x)))
```

```
(OPERATOR pick-and-free-block IS-COMPLEX
;this is a complex operator which accomplishes the setting aside of a
; block as well as de-committing it if necessary.

(GOAL           (ready(s,x))

(PRECOND        (TRUE))

(DECOMP         (FINAL SUBGOAL pick
                (setaside(s,x))

                (FINAL SUBGOAL free
                (NOT committed(x))

(EFFECTS        ))
```

```
(OPERATOR dismantle-struct IS-COMPLEX
; an operator to partially dismantle a structure
; this is the (destructive) way to make a block be the top of a structure

(GOAL           top(s,x)

(PRECOND        (TRUE)
                (STATIC in(s,x) AND NOT base(s,x)))

(DECOMP         (FINAL SUBGOAL take-off-top
                ITERATED-OVER (y: above(y,x))
                NOT in(s,y) ))

; assumes that above(x,y) is intensional predicate as follows:
;    above(x,y) IFF (on(x,y) OR THEREEXISTS Z | (on(x,z) AND above(z,y)))
; that is, x is above y if it is directly on y or if it is on a block z which is
; above y.

(EFFECTS        ) )
```

```
(OPERATOR make-arbitrary-block-available
; this operator is a generalization of remove-from-struct, which doesn't
; require that the block be at the top of the structure.

(GOAL           NOT committed(x) AND NOT in(s,x) )

(PRECOND        (TRUE)
                (STATIC in(s,x) AND on(y,x))

(DECOMP         (SUBGOAL clear-its-top
                top(s,x) )

                (FINAL SUBGOAL remove-desired-block
                NOT in(s,x) )

(EFFECTS        ) )
```