Implementing an Incremental Hierarchical Plan Recognition System[1]

M. E. Connell, K. E. Huff, V. R. Lesser

COINS Technical Report 89-82
September 1989

[1]This work was supported in part by the Air Force Systems Command, Rome Air Development Center, Griffiss Air 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).

1. Introduction

The task of incremental plan recognition can be computationally expensive, even in a simple domain. Deriving all possible meanings of a particular action, given the limited context of previous actions, can lead to an explosion of competing interpretations, many more than expected at first glance. There are at least two causes. Some interpretations have to be considered viable because unbound parameters are possibly consistent, and multiple top-level goals can be in progress simultaneously. This paper examines a set of techniques that can be used to contain the proliferation of alternative plan interpretations as soon as possible during the recognition process. These ideas are implemented in a system called GRAPPLE. GRAPPLE is a hierarchical plan recognition system used in an intelligent interface for recognizing user goals from low-level actions.

GRAPPLE recognizes a series of user actions in order of occurrence and incrementally builds reasonable interpretations for the actions. Operators in GRAPPLE are hierarchical and are assumed to be complete. (Research directed at relaxing the assumption that the plan recognizer has complete state information is described in [Huff, Dec., 1988] and [Huff, 1989].) GRAPPLE uses the hierarchy to build interpretations through linking of operators (goals of some match subgoals or preconditions of others). By recognizing the possible goals of a user and the alternative ways to achieve them, GRAPPLE identifies the rationale of a proposed action. It then uses its picture of what the user is intending to do in order to to give assistance. Information is provided to the user before a proposed action is actually put into effect. In the case where a proposed action would not be consistent with either the existing state of the world or with previously executed actions, the user is advised of the problem and can specify an alternate action.

The key problem in plan recognition is that the large number of plan derivations implied by even a few actions in a simple world requires careful and quick control of computation. GRAPPLE does aggressive checking to rule out invalid interpretations, it then applies heuristic knowledge to focus on preferred alternatives among those that appear viable. Domain knowledge which is already available is fully exploited. Syntactic and semantic checks on expressions in the operator definitions (preconditions, constraints, and goals) are used to prune out nonviable interpretations. Additional checks, based on the heuristic that the user prefers to act rationally, are used to discard interpretations. These assume that a user will not reachieve a goal already achieved, that a user tends to continue on-going plans, and that a user prefers short plans over long plans. Consequently, only reasonable or plausible interpretations of actions are constructed.

GRAPPLE has been tested in a blocks-world environment and in a simple case of a software development environment [Huff, Nov., 1988]. Domain knowledge is contained in the set of operator definitions and associated state schema; the language used for these definitions is based on classical hierarchical planning formalisms [Sacerdoti, 1977] [Wilkins, 1984] with some extensions [Huff, 1987]. The

state of the world, represented by objects and axioms involving these objects, is implemented using Knowledge Craft.[2] The recognition algorithms themselves are domain-independent. Therefore in order to consider a new or changed domain only the set of operators and the state schema need to be changed.

Plan recognition is performed automatically, without recourse to the user for information beyond the action sequence. At any given time GRAPPLE may be working with incomplete information. Variable values for higher level operators are often not yet determined. Because not all the actions have been seen, the system does not know what the future will bring. The only input to the program are the primitive actions and the user determines their order, which may be wrong. In consequence error detection is a critical objective of the system.

In this paper a simple example consisting of a sequence of actions in the blocks-world demonstrates the techniques used to control the recognition process. GRAPPLE applies checks as paths are constructed from an action to new top-level goals or pending goals. It continues to check as variables are bound, plans are extended, and action effects are simulated. Some checks are syntactic and semantic: checks for inconsistent bindings, precondition violations, and constraint violations and some checks use the heuristic that interpretations should be plausible: checks for looping plans, redundant plans and on-going plans. All of these checks are applied as soon as possible in order to quickly discard nonviable interpretations and reduce overall computation.

2. Operator Definitions

Operators in GRAPPLE are hierarchical. Those used to demonstrate the examples in this paper are given in the appendix. There are three levels of operators. At the lowest level (**stack**, **unstack**) there is no mention of structures, rather just the positions of the blocks. At the middle level (**start-struct**, **extend-struct**, **remove-top-block**, **dismantle-struct**) the concern is with which blocks are in which structures and with a block's role within a structure. At the top level (**make-red-tower**) the concern is with what type of structure exists. GRAPPLE uses the hierarchy to build interpretations.

An operator is composed of a goal, preconditions, subgoals, constraints and effects. The linking of operators (goals of some match subgoals or preconditions of others) enables the construction of interpretations. Operators are either primitive or complex. A primitive operator is an explicit user action; it has no subgoals. A complex operator has one or more subgoals and is not an explicit action. In the blocks world **stack** is a primitive operator and **make-red-tower** is complex. It is useful to look at complex operators as referring to higher level concepts and activities. The purpose of complex operators is to decompose more complicated goals into simple ones, allowing a hierarchical view of domain activities.

[2]Knowledge Craft is a trademark of Carnegie Group Incorporated.

The preconditions of an operator define the state from which the operator can legally be executed. Consequently all preconditions of an operator must be true simultaneously before any action is taken to satisfy the operator's goal or subgoals. This implies that the preconditions of a complex operator must all be true before any primitive action in the expansion of one of its subgoals begins. There are two kinds of preconditions, normal and static. A normal precondition can intentionally be satisfied by taking actions while a static precondition can not. A precondition which is not explicitly defined as static in the operator definition is understood to be normal.

By using subgoals a complex operator is decomposed into subproblems, each of which must be satisfied before the effects of the operator are achieved. For example, **make-red-tower** is a complex operator having two subgoals. It is divided into the subproblems of starting the tower and finishing it. In the case when an operator has more than one subgoal, the order in which subgoals should be satisfied is determined solely by the state of the world and the preconditions of the operators being used to satisfy the subgoals. The only restriction is that all subgoals which are labeled "final" must be true simultaneously to enable installing of the effects clause.

Constraints restrict the bindings in the operator. They must not be violated from the time the operator's preconditions are true until the time its effects are posted. Effects are the changes to the data base (the "world") which result from executing an action or from completing a complex operator. New objects can be created, attribute values can be set, new predicates can be added, and old ones deleted. Further, an effect can be made conditional on the state of the world. Not only do the effects of an operator cause its goal to be true, but often additional changes are made in the state of the world. These are side effects.

The recognition algorithms regard the goal of an operator as the main purpose for an operator's execution. The plan network which is built by matching operator goals to subgoals and normal preconditions of other operators is central to the recognition process. In particular, the purpose of an operator might be seen as the achievement of a subgoal of a second operator which might in turn be satisfying the precondition of another operator. The reason, then, of carrying out a series of actions can be to satisfy the goal of some top-level operator; a top-level operator is one whose goal does not satisfy a precondition or subgoal of any other operator.

3. The Recognition Cycle

After GRAPPLE is first initialized, an initial world state is input to the program. The actions are then processed one at a time. Before an action is actually completed and in consequence the state of the world is changed, GRAPPLE builds interpretations of the proposed action. These are used by the program to decide whether or not an action should be taken and what, if it should, its purpose would be. A context is a world view. Different contexts provide alternative world views;

each is a separate data base state. An interpretation for an action or sequence of actions is a tree of operators where top-level operator goals are linked to primitive actions. When all the variables of the linked operators in an interpretation are bound to values, the interpretation is complete. At any given time during the recognition process there might be a number of alternative interpretations for the sequence of actions, all with operator variables partially bound. When no interpretations for an action exist, the user is advised to take another action. When many exist, focusing decisions are made to pick preferable interpretations.

GRAPPLE can consider actions leading to more than one top-level goal simultaneously. However in the case where a particular action leads to two or more top-level goals at the same time only one is the "purpose" of the action in a particular interpretation and the others are side effects.

A broad outline of the plan recognition cycle is given below.

Initialize GRAPPLE: establish links between operators.

While: there is a proposed new action.

  For each active context:

    Find all valid possible interpretation paths from action to top-level goal.

    If: No interpretations in context, go to next active context.

    Else:

      For each interpretation in context:

        Make new child context and instantiate operators.

        Test extended interpretations for inconsistent bindings.

        Test for possible constraint or precondition violation.

        If interpretation invalid:

          refute context.

          go to next interpretation.

        Else:

          Assert effects of the action in the interpretation.

          Evaluate all task preconditions, constraints, and subgoals.

          Assert effects of enabled complex operators.

          Monitor constraints, looping and redundancy.

          If interpretation invalid: refute context.

          Go to next interpretation.

      When no more interpretations in context:

        Focus: prefer child contexts with extended interpretations.

        Parent context is superseded.

        Go to next active context to interpret action.

  When no more active contexts:

  Further focus: prefer child contexts where all actions are interpreted as extensions.

  If action can be interpreted:

    Establish new group of active contexts.

If action can not be interpreted in any context.

Inform user.

Restore previous group of active contexts.

Get new action.

4. Initializing GRAPPLE

GRAPPLE is initialized by establishing the links between the operators in the library. A graph showing the links between operators is shown in figure 1. For simplification the graph does not show all links. For example, **remove-top-block** also links to **unstack** through its normal precondition, (clear ?x). For each logical expression that is a subgoal or normal precondition of an operator, alternate achievers are computed. An alternate achiever is an operator whose goal, when true, achieves the expression; a set of variable mappings is determined with each alternate achiever. The operator **stack** is an alternate achiever of the subgoal (on ?top-block ?base-block) of **start-struct** and the mapping is ?x → ?top-block, ?y → ?base-block. Two things can be observed: only part of an operator's goal need match the condition for it to satisfy the expression and an operator's goal can achieve a condition in more than one way. For example, **dismantle-struct** achieves the precondition (clear ?y) of **stack** with a mapping ?base-block → ?y and a mapping ?top-block → ?y. In this case there are two alternate achievers involving **dismantle-struct**. In the current implementation of GRAPPLE, when an operator's goal only partially satisfies a subgoal or precondition of a second operator, a link is not formed.

Once links are established they are used to generate paths from primitive actions to top-level goals. When a user action is proposed and its preconditions and constraints are not violated, possible interpretations are generated. A possible interpretation is a path, linked through goals and logical expressions from the action to a top-level goal.

5. Limiting Possible Interpretations When an Action is Proposed

The strategy used in GRAPPLE is to limit the number of possible interpretations by aggressive checking to throw away those that can not be valid. GRAPPLE does not consider possible interpretations when it can predetermine that a higher level operator's preconditions or constraints are violated, that an already satisfied goal will be resatisfied or that endless cycles exist. Checking starts as the interpretations are generated; the viable interpretations are then instantiated and checked further as described in section 6. In the identification of all possible paths from the action to the top-level goal an interpretation is discarded if any of the conditions listed below are violated. These conditions are tested as each operator is linked up-

ward. The values of the variables used in the tests are those given as parameters to the action; these values are propagated up through the links. For example, the action (unstack cube3 cube1) leads to **dismantle-struct** with ?top-block=cube3 and ?base-block=cube1. The construction of a particular path upward is discontinued as soon as any operator fails the tested conditions. The result is that the computation involved in generating possible interpretations, and the number of these interpretations, is greatly reduced.

Tested syntactical and semantic conditions in the bottom-up generation of possible paths:

- If an operator is linked, either directly or indirectly, to the action through one of its subgoals then none of its preconditions can be false. The truth of a precondition is determined by querying the data base. If all the variables in the precondition are bound to values, the precondition must be true. For example, if in the initial state cube3 is on cube1, then the goal of the action, (unstack cube3 cube1) satisfies the subgoal of **dismantle-struct**, the variable mapping is cube3 → ?top-block and cube1 → ?base-block. If any of the preconditions of **dismantle-struct** are false with this variable mapping, such as cube3 not being in a structure, then no possible path can be derived which starts by **unstack** linking to **dismantle-struct** in this way.
- No constraints of an operator are violated. Again the bindings are those taken from the action (as propagated up through the path link via the mappings). Since there are no constraints on **dismantle-struct**, the test will not apply. If cube1 eventually maps into ?first-cube of **make-red-tower**, then it must be red to satisfy the top-level constraint, (color ?first-cube red).

Tested conditions, based on the heuristic that the user prefers to be rational, in the bottom-up generation of possible paths.

- No goal of an operator having all its variable values bound from the parameters of the action when propagated upward is already true in the current state of the world. The user's action in such a case would serve no purpose since the intent of the action is seen as satisfying an already satisfied goal. If a goal is already true, any interpretation which reachieves it should be discarded.
- No linked precondition or subgoal of an operator is already true in the current data base state. For example the path starting as follows (unstack cube3 cube1) leading to the subgoal (not (on ?top-block ?base-block)) of **dismantle-struct** leading to the precondition (clear ?y) of **stack** with cube3 = ?top-block = ?y will be discarded because cube3 is already clear in the current state.
- No loops exist in the interpretation. No operator is revisited on a path with identical variables bound to the same values or with identical variables unbound. This insures that possible interpretations in which cycles occur are not considered. For example, the operator (stack cube2 ?y) can not occur more than once in a path, although both (stack ?x ?y) and (stack cube2 ?y) could appear.

In order to inhibit endless searching a given cutoff level or maximum number of operator links allowed is input to GRAPPLE. This is based on another "rationality" heuristic: that the user will not prefer long paths to top-level goals when shorter ones exist. There are some instances when an action taken in a given data base state can not lead to any top-level goal through a reasonable number of links. When this occurs or when there are no valid possible interpretations, the action can not be recognized and the user is so advised.

6. Exploiting Available Constraints

The process of limiting the derivation of interpretations and checking those that remain is best understood by looking at a particular example. The illustrations in the figures are actual output from the program. Figure 2 shows the initial state of the world for the example. There is only one context, the initial world state, in which to interpret the first action. The first action is (unstack cube3 cube1). Four possible interpretations leading to the goal of completing a red tower are derived. Nineteen possible interpretations are discarded while testing the conditions (described in section 5) in the process of finding these potentially acceptable paths.

The four interpretations are instantiated before further analysis is done to see if there are additional reasons to discard them. Each interpretation is tested in turn. For each, the original parent context is copied to a new context. Every operator of the interpretation within the new context is instantiated as a separate task (instantiated operator) and its preconditions and subgoals are expanded. The original parent context, replaced by new ones, is *superseded.* When an interpretation within a context is found to be invalid, the context is *refuted.* In the example, there are four ways to interpret the first action in the parent context ; four new contexts are created. Figure 3 shows the context tree after the instantiation of the four interpretations of the first action. Figure 4 shows the instantiation of an interpretation in a particular context.

As stated previously, preconditions of operators linked to the action through a subgoal must be true for the interpretation to be valid. No interpretations with operators having false preconditions and bound variables survived the derivation, but there may exist operator preconditions with unbound variables which are false because values do not exist for those variables in the existing data base state which will make the expressions true. Because it is necessary that all preconditions of a complex operator be true before any action is taken which leads to one of its subgoals, the program allows the binding of variables through preconditions from the existing state of the data base when such an action is proposed. Any variable values which are found in this way must not violate the constraints of the operator. If a variable is propagated up through links of the interpretation and is included in other operator constraints or preconditions which must also be true (when the operator is linked through a subgoal), then these expressions further the restrictions on the bindings. The result is a query to the data base with a set of logical expressions

which must all be true simultaneously. This query constrains the search for plans in three ways. It throws out invalid interpretations, it binds variables when the bindings are unique, and it provides constraints on variables when the bindings are not unique.

In the first interpretation of the example (unstack cube3 cube1) satisfies the subgoal of **dismantle-struct** (all its preconditions are true with bound variables from the action); **dismantle-struct** satisfies the normal precondition, (on-table cube1) (clear cube1), of **stack**; **stack** leads to the subgoal (on cube1 ?base-block) of **start-struct**; and **start-struct** leads to a subgoal of **make-red-tower**. In the solution all the preconditions of **start-struct** must be true for the interpretation to be valid. The data base is queried with the two precondition clauses of **start-struct** and the relevant constraint clauses of **make-red-tower**. As a result the variable ?base-block of **start-struct** is bound to cube2 from the data-base; this is the only value in the data-base which is on the table, not equal to cube1 and satisfies the constraints (?first-cube being a cube and red) of the operator **make-red-tower**.

When there are no values for the variables in the data base which satisfy the query, the precondition is false. This is what happens in the fourth interpretation which is instantiated in context5 (see figure 3). The path from the action, (unstack cube3 cube1), is **unstack** → **dismantle-struct** → **stack** → **extend-struct** → **make-red-tower**. **Extend-struct** leads to the subgoal (add-pyramid) of **make-red-tower**. The variables ?top-block and ?base-block of **extend-struct** are both unbound. The data base is queried using the preconditions of **extend-struct** and relevant constraints of **make-red-tower**. The query fails because the expressions can not be solved simultaneously. This is because ?lower-block and ?base-block both are bound to cube1 which violates the precondition that they can not be equal. The interpretation is discarded and the context refuted. Notice context5 in figure 3 is refuted. When more than one value can be bound to a variable, the alternative possible values of the variable are added as a constraint on the task. The effects of any task which affect or use such a variable are not activated until one of the multiple values is actually forced on the interpretation by a subsequent action. The additional constraint on the variable may restrict the possible extensions of the interpretation.

By exploiting the constraints in the example, GRAPPLE determined that there were only three possible interpretations for the proposed first action.

Finally it should be emphasized that GRAPPLE binds variables in only three ways: from an overt action and the linking of operators, from asserting effects which create new variable values, and from a query to the data base with expressions which must be true.

7. Monitoring Constraints, Looping, and Redundancy with Data Base Changes

Some interpretations may be discovered to be invalid only after the effects of the action are simulated. For instance, a side effect of an operator could cause the violation of a precondition or constraint of another operator (upward on the path).

The state of the world actually changes in each *active* (neither *superseded* or *refuted*) context when the effects of an action or operator are asserted. (In GRAPPLE all the contexts which are active are carried forward.) Afterwards every task (instantiated operator) in the context is then processed to see if previously unsatisfied preconditions or subgoals are now satisfied due to the data base changes. Tasks exist in varying stages of completion - :trying-to-achieve- preconditions, :trying-to-achieve-subgoals, or :accomplished. When all of the tasks in a context are accomplished, the top-level goal is achieved. If the state of a task is :trying to achieve-preconditions, the preconditions having bound variables are tested. When all of a task's preconditions are true simultaneously, the state of the task is changed to :trying-to-achieve-subgoals. If the state of a task is :trying-to-achieve-subgoals, the subgoals having bound variables are tested. Once the subgoals are satisfied and the truth of all the task constraints on the interpretation path is established, the effects of the task are asserted. The effects of complex operators are automatically asserted when its subgoals become true and that primitive operators are overt actions which must be taken by the user.

Redundant plans or actions and looping may be discovered as variables are bound. One of the ways a context can be refuted is by the discovery that there exists two different tasks in the context which are instantiations of the same operator and which either have matching bindings or have the set of bindings for one task subsumed by the bindings of the other. This means that these two tasks are combined in some other context to form one interpretation or that looping occurs. Notice that it is possible that a precondition or subgoal can become true as a result of asserting the effects of a task even though the goal of the task in the interpretation is not to satisfy that particular condition. No variables are bound during the process of checking the truth of preconditions or subgoals of a task; an expression is true only if its variables are bound.

There is an additional reason to discard a context when the data base is altered. Changes in the world state may violate operator constraints. Therefore, when any constraint is found to be false while processing the tasks, the context is refuted.

Consider the example. There are three active contexts for the first action after the effects are posted in each (see figure 3). The interpretation in context2 is illustrated in figure 4. After the action (unstack cube3 cube1) the tasks unstack3 and dismantle-struct3 are completed; and the preconditions for stack3 are true. The task, start-struct3, is in the state :trying-to-achieve-subgoals as is the task make-red-tower3. Three subgoals need to be satisfied in order to achieve the top-

level goal. If all the child contexts (context2, context3, context4) are refuted the primitive action can not be recognized and the user is is so advised. This does not happen in the example. In the example none of the simulated effects of the first action caused GRAPPLE to discard an interpretation. The action is taken in each active context.

8. Narrowing Interpretations with Additional Actions

Additional actions reveal more of the plan and provide information which is used to discard existing interpretations.

After the first action is processed in all active contexts, GRAPPLE asks the user for the next action. This action is considered in each active context in turn. In the example (stack cubel cube2) is the second action; it is interpreted in context2, context3, and context4. Possible interpretation paths are derived and discarded as before. There is a difference though and that is: a path can lead to an operator corresponding to an instantiated task which is pending (not yet complete) in the context as well as to a top-level operator. For example: since stack3 is a pending task in context2, a possible interpretation leads to **stack**. In each context of the example four possible interpretations out of eight survived the derivation.

Each surviving candidate interpretation can either be an extension of an existing interpretation or, if it leads to a top-level goal, the start of a new interpretation in the given context. An interpretation which does not lead to a top-level operator must necessarily be an extension. An interpretation which does lead to a top-level operator might be an extension if it leads to an unsatisfied precondition or subgoal of a partially satisfied top-level task. Alternatively it might be the start of a new interpretation to a top-level goal.

There may be reasons to reject an extended interpretation before tasks are instantiated. When the bindings of the new action are inconsistent with the existing bindings of the top-level task, when the new bindings will cause a constraint violation in the existing interpretation or when a precondition of a task linked through its subgoal is violated with the new bindings added, the interpretation is not considered as an extension. For example, in both context4 and context3 two of the possible interpretations, one leading to **stack** and another leading to **start-struct** are thrown out due to inconsistent bindings.

For each remaining candidate interpretation, whether it is an extension or a new interpretation, the active context is copied to a new context along with the pending tasks. The parent context is superseded. The analysis done in each surviving context is the same as is described for the first action. Contexts are refuted if necessary preconditions can't be true. Pending tasks are processed, effects are asserted and further tests are done to refute contexts.

In building new interpretations, there is a chance that redundant plans evolve. In cases where an interpretation can be viewed as both an extension and the start

of a new interpretation, a test is done to see if the two interpretations are in fact the same. If the top-level goals are identical and the variable bindings of the new interpretation are subsumed by those of the extension, the context corresponding to the start of a new interpretation is refuted. This is what happens when the final action of the example is interpreted as the start of a new plan in context9 (described in the next section).

Looking at the example, the action (stack cube1 cube2) can be interpreted as an extension of the existing interpretation in context2. The bindings are consistent and no constraints or necessary preconditions are violated. Those interpretations where bindings are inconsistent, constraints are violated or preconditions violated are not pursued. The effects of **stack** are asserted, accomplishing stack11 and start-struct10 and achieving a subgoal of make-red-tower10. Figure 7 shows the task tree for the interpretation. Of the possible interpretations of the action in context2 three child contexts, context10, context11, and context12 are refuted because two instantiations of the operator **stack** exist with identical bindings in the context. Figure 6 represents the context tree after the second action. In context3 the action (stack cube1 cube2) can not be interpreted as an extension because the bindings are inconsistent. The only interpretation allowed in this context is one leading to a new top-level goal of making a red tower.

As it stands the program does not resolve the problem which arises when achieved preconditions or subgoals of a task are violated prematurely by asserting the effects of later actions. In this case a decision should be made whether to go ahead with the action or retract it. If an action is retracted, it must be retracted in all contexts. On the other hand when the action is taken, previously satisfied conditions which are violated must be resatisfied by future actions. The existence of conditions in a context that are violated in this way could also be used to focus the recognizer. The focusing heuristic is: the user prefers plans in which conditions do not have to be resatisfied.

9. Focusing on Extended Interpretations

Another way of controlling the plan recognition process is to focus on selected interpretations while saving others for future development. When a new user action is proposed, GRAPPLE first focuses on interpretations where the action extends an existing plan. The program prefers to assume that the user has a plan or set of plans in mind [Carver, 1984]. Unrefuted contexts in which an action can be interpreted as extending an existing plan are made active; other unrefuted contexts are made inactive. An *inactive* context, neither refuted or superseded, is saved for future processing. In the example, context8 is made inactive because (stack cube1 cube2) can not be interpreted as an extension, but it can be interpreted as the start of a new plan. When there are no contexts in which the action can extend a plan, all contexts where the action starts a new plan are made active.

When there are no contexts in which an action can be interpreted, the action can not be recognized. Consider the situation shown in figure 5. The second action of the example has been processed. If the user then proposes an action such as (stack pyrl cube3), GRAPPLE can not find a way to interpret the action in any active context. It then makes all inactive contexts active, processes the unprocessed actions in the newly activated contexts and tries again to interpret the given action. If the action still can not be recognized, the user is informed. The interpretations and contexts are then changed back to the state that existed before the original action was proposed. The above scenario is not illustrated in the example.

The third action of the example is (stack pyrl cube1); it is processed in the active contexts , context9 and context7 and is added to the list of unprocessed actions in the inactive contexts, context8 and context6. All the candidate interpretations in context7 are found to be invalid. The action, interpreted as an extension, in context9 achieves the subgoal of extend-struct3 which achieves the second subgoal of make-red-tower17. The action, interpreted as the start of a new plan, in context9 gives a redundant interpretation. The top-level goal is accomplished in context14. The final context tree is shown in figure 9 and the final interpretation in figure 10. The plan is recognized.

10. Multiple Plans

The ability to limit possibilities is even more valuable when there are multiple plans being carried out in parallel. GRAPPLE can recognize more than one plan simultaneously. When a top-level operator, **make-blue-tower**, is added to the set of operators, actions for building a red tower and a blue tower can be interleaved and the program recognizes the purpose of the actions. For instance, the first action of unstacking a blue cube from a red cube will have an additional interpretation of dismantling the original structure in order to build a blue tower.

11. Conclusion

We have found that incremental hierarchical plan recognition is more computationally expensive than expected. Even in a simple domain the task can quickly get out of control.

In a simple example nineteen possible interpretations for the first action are reduced to three (of these twelve are discarded based on the heuristic that the user will behave rationally and interpretations are plausible). Checking and focusing reduces the number of interpretations for the first followed by the second action from twenty-four to two (of these eleven are discarded based on the "rationality" heuristic). Finally GRAPPLE finds only one way to interpret the sequence of three actions out of ten possible interpretations (of these two are discarded based on the "rationality" heuristic).

Non-trivial mechanisms are required to control the combinatorial explosion of

possible interpretations. There is a need to exploit constraints quickly and to focus on reasonable or plausible interpretations. With the techniques described here incremental plan recognition becomes tractable.

Appendix
Example Blocks World Operators

unstack

```
(defplan unstack (?x ?y)
(goal (not (on ?x ?y)))
(precondition on-x-y (on ?x ?y) :static)
(precondition clear-x (clear ?x))
(precondition not-eq (not (= ?x ?y)) :static)
(subgoal (primitive))
(effects (add (clear ?y))
         (add (on-table ?x))
         (delete (on ?x ?y))))
```

remove-top-block

```
(defplan remove-top-block (?top-block ?block2 ?structure)
(goal (and (clear ?top-block) (on-table ?top-block)
         (clear ?block2)))
(precondition not-on-table (not (on-table ?block2)) :static)
(precondition in-struct1 (in ?block2 ?structure) :static)
(precondition in-struct2 (and (in ?top-block ?structure)
         (top ?structure ?top-block)) :static)
(precondition on-blk (on ?top-block ?block2) :static)
(precondition not-eq (not (= ?top-block ?block2)) :static)
(subgoal unstack-blks (not (on ?top-block ?block2)) :final)
(effects (delete (top ?structure ?top-block))
         (delete (in ?top-block ?structure))
         (add (if (old (not (base-block ?structure ?block2)))
                  THEN (top ?structure ?block2)))
         (delete (if (old (base-block ?structure ?block2))
                  THEN (in ?block2 ?structure)))
         (delete (if (old (base-block ?structure ?block2))
                  THEN (base-block ?structure ?block2)))
         (set (type-struct ?structure unknown))))
```

dismantle-struct

```
(defplan dismantle-struct (?top-block ?base-block ?structure)
(goal (and (clear ?base-block) (on-table ?base-block)
         (clear ?top-block) (on-table ?top-block)))
(precondition in-struct1 (and (in ?base-block ?structure)
         (base-block ?structure ?base-block)) :static)
(precondition in-struct2 (and (in ?top-block ?structure)
         (top ?structure ?top-block)) :static)
(precondition on-blk (on ?top-block ?base-block) :static)
(precondition not-eq (not (= ?top-block ?base-block)) :static)
(subgoal unstack-blks (not (on ?top-block ?base-block)) :final)
(effects (delete (top ?structure ?top-block))
         (delete (in ?top-block ?structure))
         (delete (in ?base-block ?structure))
         (delete (base-block ?structure ?base-block))
         (set (type-struct ?structure unknown))))
```

stack

```
(defplan stack (?x ?y)
(goal (on ?x ?y))
(precondition not-com3 (and (on-table ?x) (clear ?x)))
(precondition not-pyr-y (not (pyramid ?y)) :static)
(precondition clear-y (clear ?y))
(precondition not-eq (not (= ?x ?y)) :static)
(subgoal (primitive))
(effects (add (on ?x ?y))
         (delete (clear ?y))
         (delete (on-table ?x))))
```

start-struct

```
(defplan start-struct (?base-block ?top-block ?structure)
(goal (and (on ?top-block ?base-block)
           (top ?structure ?top-block)
           (base-block ?structure ?base-block)
           (in ?top-block ?structure)))
(precondition on-table (on-table ?base-block) :static)
(precondition not-eq (not (= ?base-block ?top-block)) :static)
(subgoal stack-blks (on ?top-block ?base-block) :final)
(effects (new (?structure structure))
         (add (top ?structure ?top-block))
         (add (in ?top-block ?structure))
         (add (in ?base-block ?structure))
         (add (base-block ?structure ?base-block))
         (set (type-struct ?structure unknown))))
```

extend-struct

```
(defplan extend-struct (?lower-block ?top-block ?base-block ?structure)
(goal (and (on ?top-block ?lower-block)
           (top ?structure ?top-block)
           (in ?top-block ?structure)
           (base-block ?structure ?base-block)))
(precondition in-struct (in ?lower-block ?structure) :static)
(precondition in-struct1 (base-block ?structure ?base-block) :static)
(precondition not-eq1 (not (= ?base-block ?top-block)) :static)
(precondition not-eq2 (not (= ?lower-block ?base-block)) :static)
(precondition not-eq (not (= ?lower-block ?top-block)) :static)
(subgoal stack-blks (on ?top-block ?lower-block) :final)
(effects (add (top ?structure ?top-block))
         (add (in ?top-block ?structure))
         (delete (top ?structure ?lower-block))
         (set (type-struct ?structure unknown))))
```

make-red-tower

```
(defplan make-red-tower (?structure ?first-cube ?second-cube ?pyramid)
(goal (redtower ?structure))
(subgoal build-foundation (and (on ?second-cube ?first-cube)
         (in ?second-cube ?structure)
         (base-block ?structure ?first-cube)) :final t)
(subgoal add-pyramid (and (in ?pyramid ?structure)
```

```
                    (base-block ?structure ?first-cube)
                    (on ?pyramid ?second-cube)) :final t)
(constraints (pyramid ?pyramid) (cube ?first-cube) (cube ?second-cube)
                    (color ?pyramid red) (color ?first-cube red)
                    (color ?second-cube red))
(effects
                    (set (type-struct ?structure tower))
                    (set (color ?structure red))))
```

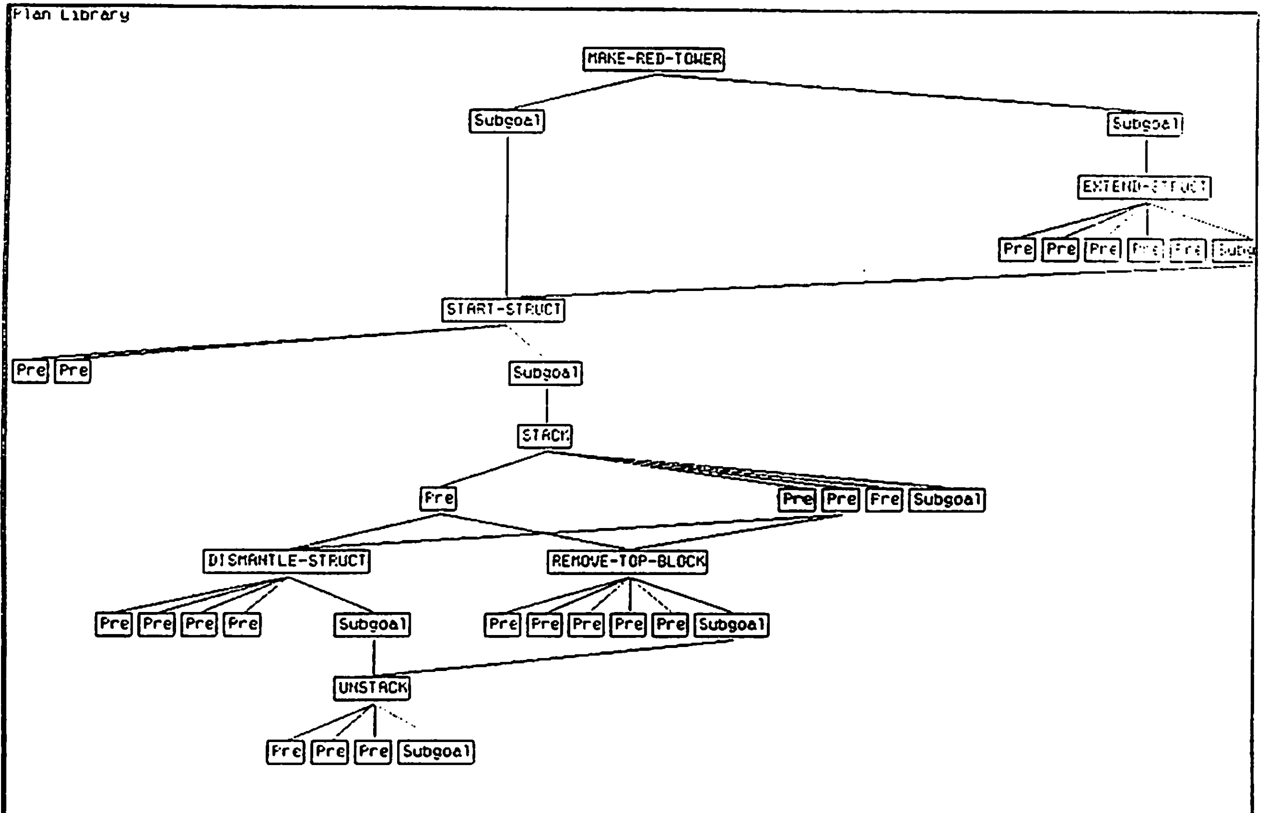

Figure 1: The Plan Library

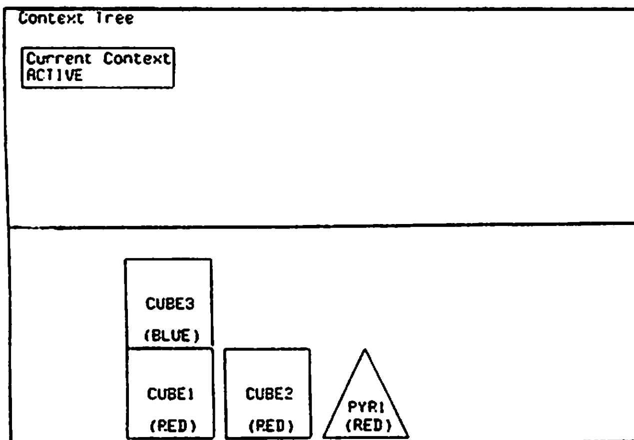

Figure 2: The Initial State and Context

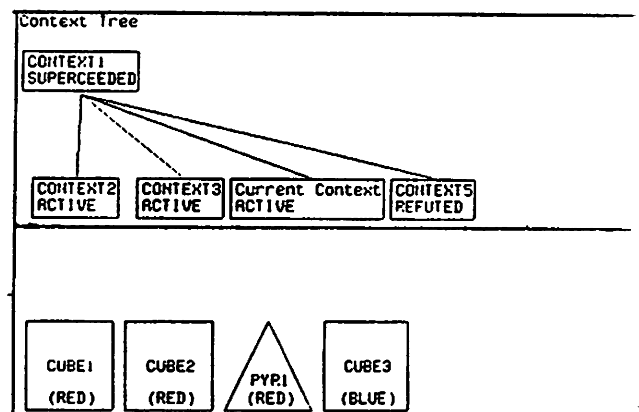

Figure 3: State and Contexts after the actio
(unstack cube3 cube1)

Figure 4: Interpretation in Context2 after the action

(unstack cube3 cube1)

Figure 5: State after the action (stack cube1 cube2)

Figure 6: Contexts after the action (stack cube1 cube2)

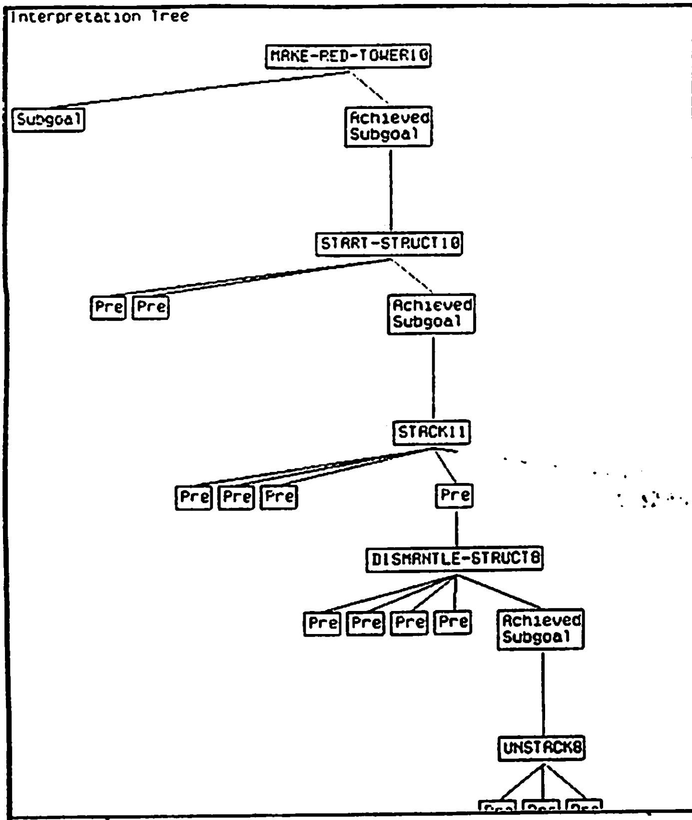

Figure 7: Interpretation in Context9 after the action (stack cube1 cube2)

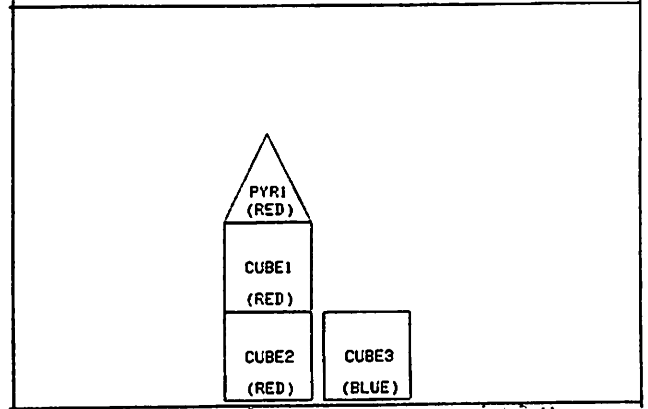

Figure 8: State after the action (stack pyr1 cube1

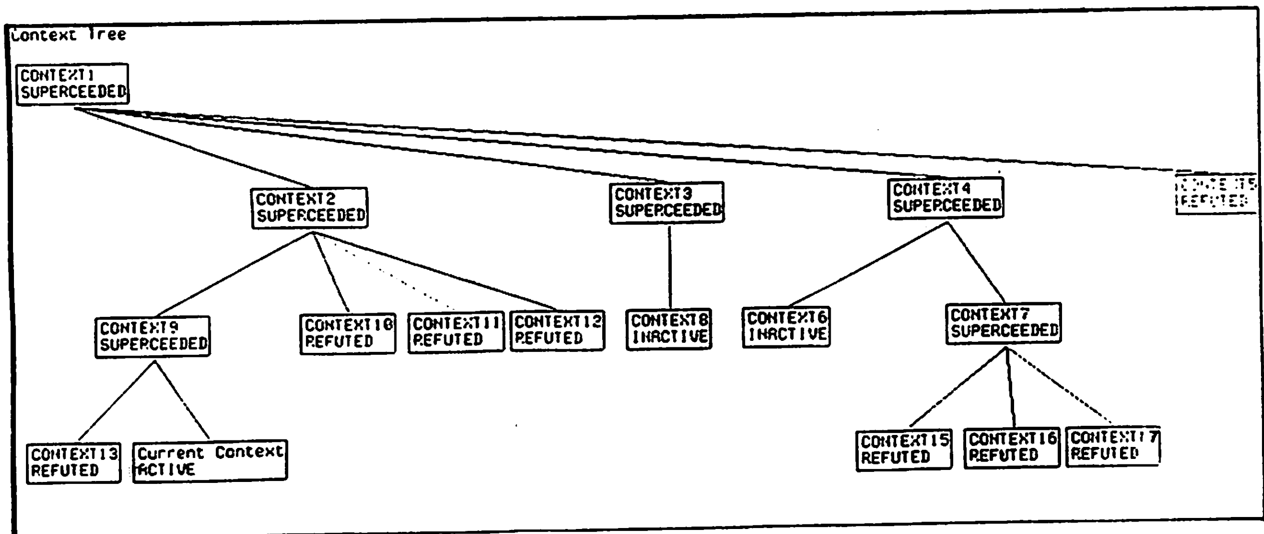

Figure 9: contexts after the action (stack pyr1 cube1)

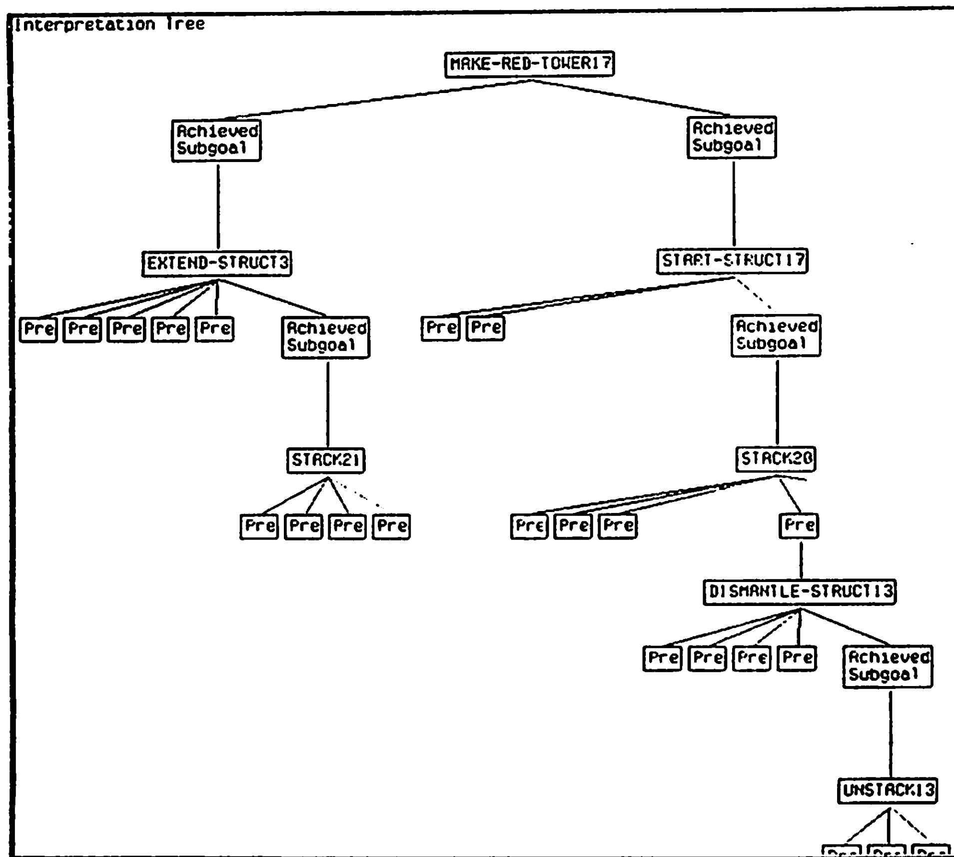

Figure 10: Interpretation in Context14 after the action (stack pyr1 cube1)

References

Carver, N.F., Lesser V.R., McCue, D.L. "Focusing In Plan Recognition" in *Proceedings of the National Conference on Artificial Intelligence* Austin, Texas, 1984.

Huff, K.E., Lesser V.R. "A Plan-based Intelligent Assistant That Supports the Software Development Process" in *Proceedings of the Third Symposium on Software Development Environments* ACM, Boston, November, 1988.

Huff, K.E., Lesser V.R. *The GRAPPLE Plan Formalism* COINS Technical Report 87-08: University of Massachusetts, Amherst, MA. 1987.

Huff, K.E., Lesser V.R. *Plan Recognition in Open Worlds* COINS Technical Report 88-18: University of Massachusetts, Amherst, MA. Dec., 1988.

Huff, K.E. *Plan-based Intelligent Assistance: An Approach to Supporting the Software Development Process* Ph.D dissertation, University of Massachusetts, September, 1989.

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

Wilkins, D.E. "Domain-Independent Planning; Representation and Plan Generation." Artificial Intelligence, 22 (1984), 269-301.