Planning in The Context of an Intelligent Assistant[1]

M. E. Connell, K. E. Huff, V. R. Lesser
Computer and Information Science Department
University of Massachusetts

COINS Technical Report 91-44
May 1991

Abstract: This paper describes an intelligent assistant that works cooperatively with a user, incrementally recognizing user plans and planning for him when needed. The system integrates planning and plan recognition. This paper concentrates on the assistant's planning component. We have developed methods to control search using tests on constraints embedded in the hierarchical domain operators and using heuristic rules to focus the assistant on plausible interpretations and reasonable plans. Although the system is domain independent we have applied it specifically to software developement.

[1]This work was supported by University Research Initiative grant number N00014-86-k-0764.

1 Introduction

A person who is solving a problem with the help of an assistant expects to be able to take the initiative, to have mistakes corrected, and to get good advice. An intelligent system which can assist a user in these ways is a valuable assistant. Effective cooperation comes about when an assistant can recognize user plans and can plan for him as needed. The intelligent assistant is an integrated system of planning and plan recognition.

In order for the assistant to understand the rationale for user actions it must have access to domain knowledge. This knowledge is stored in hierarchical domain operators and in a set of predicates, representing the state of the world. This knowledge may not completely describe the world. The system therefore includes supplemental information, consisting of assumptions about the world derived from experience.

This paper presents a system which incrementally builds a model that explains user actions, enabling it to critique further actions, summarize work that has been done, establish agendas of tasks to do, and plan automatically. The difficulty in doing this is that there may be many competing ways to interpret actions and satisfy goals. We have developed methods to control search using tests on constraints embedded in the hierarchical domain operators and using heuristic rules to focus the system on plausible interpretations and reasonable plans. This paper concentrates primarily on the system's planning component.

The recognition component of the system gives assistance by responding to user actions. Because a person usually prefers to decide for himself whether or not he needs help, the user is given the choice of proposing an action or of deferring to the assistant. When the user takes the initiative, the recognizer looks for errors and gives advice when it finds them. This requires incrementally building interpretations for actions in the order that they occur which makes it possible to determine the possible goals of the user. Having identified the rationale of the proposed action, the system gives advice using its picture of what the user is intending to do. 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. When there are many possible interpretations for the proposed action, heuristic rules recommend one.

The planning component of the system gives assistance by proposing actions to the user. When the user defers to the assistant, the system generates actions which will eventually complete user-specified goals. Obviously to work effectively with the assistant the user must be familiar with the operator definitions and understand formulated goals. A user, informed to this extent, can employ the assistant to determine the status of goals and can then indicate the goal or goals he wants satisfied. The system then can plan.

The key problem in both planning and recognition is that computation can become expensive. In plan recognition a large number of plan derivations follow from even a few actions. Deriving all the possible meanings of a particular action, given the context of previous actions, can lead to an explosion of competing interpretations.

Similar problems occur in planning. There may be many possible ways to complete a goal, many alternative action bindings and action sequences. A set of techniques used to select viable interpretations during recognition is described in earlier work [Huff, 1989] and [Huff, 1988a]. In this paper we extend these techniques and develop a set of heuristics for choosing appropriate operators and variable bindings, allowing the intelligent assistant to plan efficiently.

The ideas are implemented in a system called GRAPPLE [Huff, 1987], [Huff, 1988a]. Originally it was an incremental hierarchical plan recognition system, tested in the blocks-world and software developement domain. Mechanisms for controlling the number of possible interpretations and choosing the most viable are described in earlier papers [Connell,Huff,Lesser, 1989] [Huff, 1988a]. Here, we extend the implementation of GRAPPLE to include incremental planning and discuss the issues involved.

In other work where planning and plan recognition have been used to provide automatic assistance there has been emphasis on analyzing misconceptions or bugs in the user's plan [Genesereth, 1979] [Miller and Goldstein 1977] [Croft, 1988]. When a user's action fails to meet the planner's expectations, in these systems, the reason for failure is investigated; a library of errors or bug types is consulted or an exception is classified and repair is suggested. A particular example is POLYMER [Croft, 1988], another interactive planning system which works in cooperation with the user. In this system the expansion of a goal, specified by a user, leads to the construction of a procedural net and a set of expectations for the next step in the plan. When the user's action doesn't match the system's expectations the user and the system can negotiate. One of the options at this point is to backtrack and try an alternate expansion. Search is limited by built in temporal orderings on goals, tasks and actions. In comparison, it is the recognizer in GRAPPLE that finds errors; the user can't negotiate. The planner in GRAPPLE is invoked only when the user asks for help in which case all possible next plan actions are derived and ranked. The best action is suggested to the user. POLYMER allows the user to have more control over the development of the plan than GRAPPLE does. This may require more thinking on the part of the user.

The intelligent assistant, presented in this paper, detects user mistakes. However, no effort is expended in analyzing, classifying or understanding them. One can retract the proposed, mistaken, action and either take a new one or ask the assistant to do so. Automatic planning in GRAPPLE can be invoked anytime when help is needed, in response to a detected error and in rescuing a muddled user.

The paper is organized as follows. Section 2 presents an overview of the system, section 3 discusses the issues involved in controlling search, and section 4 describes the domain knowledge and operator definitions. The following sections, 5, 6, and 7, present the contributions of three kinds of knowledge (domain knowledge, empirical knowledge and heuristic knowledge) to the control of search. Section 8 describes an example session with the assistant and section 9 concludes.

2 Overview of the System

The extended GRAPPLE system has been tested with several sets of operators and state schema in the software developement domain. For instance in one example the assistant supports the user in building, testing and archiving a system by providing independent judgment on user actions, summarizing what has gone on, and suggesting actions. (The operators relevant to this world are given in the appendix.) This automatic assistance is particularly helpful in complex domains where humans are prone to making mistakes on details.

If the user takes the initiative and proposes an action, the system builds interpretations for the action and, using a set of heuristics, focuses on one as preferred. When no interpretations are found, the user is advised to take another action. If the user wants help, he can find out from the assistant the state of the world and the state of goals within the current context. This information allows him to work with the assistant; he can specify a goal and the system will determine the "best" action or sequence of actions which achieve that goal; he can specify a number of goals and the system will determine the "best" goal to satisfy and the "best" action that achieves it. Interaction between assistant and user is illustrated in an example session shown in Section 8. In this session the user's task is to build a system from the existing baseline. At the point where he proposes to archive the system without running the existing tests, he is advised to take another action. As a result he asks for help and the assistant suggests that a relevant test case be run. After a proposed action is taken the state of the world and the interpretation are updated. The user can then choose to propose a new action or specify another goal.

It should be emphasized that GRAPPLE demonstrates the intelligent assistant in a simulated world. The issues involved in incorporating the system into a working environment are not investigated in this work.

3 Controlling Search in Incremental Planning

The overall objective in planning is to find a credible, correct and short sequence of actions to achieve the user-specified goals. When there is no predetermined commitment to the ordering of actions, it may be necessary to test the correctness of many alternatives. Finding a solution by simulating the effects of actions, predicting future states of the world, and reflecting on these successive world states is time-consuming and computationally expensive. We have developed a set of techniques for ordering goals, and for choosing actions and action bindings which address these problems.

Ideally, a planner should determine a successful and simple plan in a short amount of time. To do this, search should be efficient. GRAPPLE exploits three kinds of knowledge to control search.

The first kind of knowledge is domain knowledge embedded in the hierarchical operator definitions. The system uses the hierarchy to build, incrementally, an interpre-

tation for a partial sequence of actions. (An interpretation for an action or sequence of actions is a tree of instantiated operators where top-level goals are linked along a path to primitive actions.) An evolving picture of the rationale for actions already tested guides the planner's search for new actions. Syntactic rules govern allowable actions while variable bindings are determined through ongoing checks of operator constraints. With each new action the interpretation is expanded; the new interpretation can then be used to guide search for the next action. This phenomenon is illustrated in the example.

The second kind of knowledge is empirical, explicitly represented in the system. It is used to make inferences about non-observable conditions in the world. Actions which are consistent with what is known about the state of the world as well as consistent with what is assumed about the state of the world are preferred actions. For instance one would rather run a test case that is assumed to be applicable to the system than run a case that is assumed not to be. Further one would prefer to run the case that is known to be relevant than one that is assumed to be.

The third kind of knowledge is contained in a set of heuristic rules. These are used to narrow search by making choices among goals and actions. Once a user designates a condition (this may be a precondition or subgoal of an incomplete higher level operator) or set of conditions he wishes to complete, the planner searches for the actions which satisfy or start to satisfy the conditions. It returns a list of all possible primitive actions along with variable bindings. Each either satisfies the goal or is the first in a sequence of actions that satisfies the goal. The heuristic rules are used for choosing the best action and variable bindings from this list. One rule,for example, states that a chosen action should either directly satisfy a goal or reach it in a small number of path links. The purpose of these rules is to keep the plan simple and credible and to avoid mistakes and replanning.

4 Domain Knowledge in Operator definitions

Domain knowledge in GRAPPLE is contained in the set of hierarchical operator definitions and associated state schema; the language used for these definitions is based on classical planning formalisms [Sacerdoti, 1977] [Wilkins, 1984]. The state of the world is represented by objects and axioms involving these objects. The planning and plan recognition algorithms themselves are domain-independent. Therefore in order to consider a new or changed domain only the set of operators and state schema need to be changed.

An operator is composed of a goal, preconditions, subgoals, constraints and effects. Operators are either primitive or complex. A primitive operator is an explicit action; it has no subgoals. A complex operator has subgoals and is not an explicit action. GRAPPLE uses the hierarchy to build interpretations through linking of operators; goals of some match subgoals or preconditions of others. In the appendix there is an example of a set of operators. The operator **build** is complex and the operator **archive**

is primitive (see Figure 1).

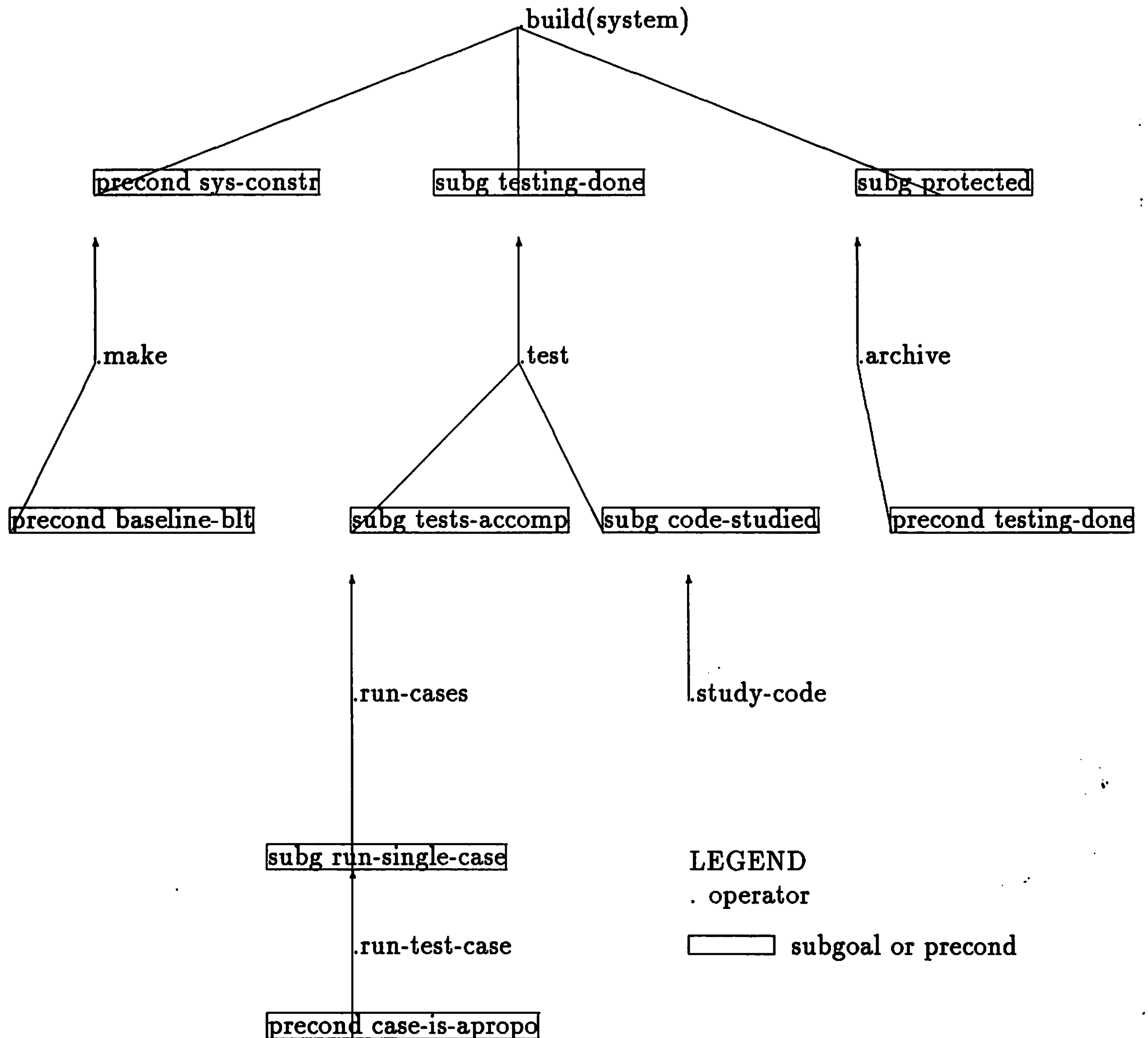
The preconditions of an operator define the state from which it can legally be executed. All preconditions of an action must be true simultaneously before it can be executed. The preconditions of a complex operator must all be true before any primitive action in the expansion of one of its subgoals begins. There are two kinds of preconditions. A normal ('notstatic') precondition can intentionally be satisfied by taking actions while a static precondition can not. When not all the preconditions of an operator are satisfied, those that are are protected.

By using subgoals a complex operator is decomposed into subproblems, each of which must be satisfied before the effects of the operator are posted. 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. A subgoal of an operator is protected when it has been satisfied and some other subgoals of the operator are waiting to be satisfied.

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. Not only do the effects of an operator cause its goal to be satisfied, but often additional changes are made in the state of the world. These are side effects.

The goal of an operator is the main purpose of the operator's execution. The plan network which is built by matching operator goals to subgoals and normal preconditions of other operators is, as already described, central to the planning process. Figure 1 shows the links between operators in the example plan library. In particular, the purpose of carrying out an operator might be the achievement of a subgoal of a second operator which might in turn be satisfying the precondition of another operator. The reason for taking 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.

Figure 1 The Plan Library



Consider the top-level operator **build**, given in the appendix. It is complex with two subgoals, testing-done and status-protected, and a nonstatic precondition, sys-constructed. For this particular operator there are no constraints.

5 Empirical Knowledge

The intelligent assistant does not assume that information contained in the operator definitions is complete. The system allows assumptions to be made about the state of

the world. Inferences about non-observable conditions are made on the basis of empirical knowledge and on the basis of what can be observed. This empirical knowledge is expressed as rules in a Truth Maintenance System (TMS) [Doyle, 1980]. It is fully described in [Huff, 1988b]. These inferred conditions are given values, either true for certain, true by assumption, false for certain, or false by assumption. The value is the result of evaluating a TMS rule with a set of action-variable bindings. A static precondition, constraint, or subgoal of an operator can be based on inferences from the observable world. For instance the action **run-test-case** has a static precondition that the case is applicable to the system. The state of this precondition is not observed directly; it is inferred. The use of assumptions about non-observable conditions makes some actions and variable bindings more credible than others. An action which can be interpreted without making any assumptions about conditions in the world is preferred. An action which can be interpreted without changing assumptions about the world is a better alternative than one which causes current assumptions to be violated or altered. A good plan doesn't force the system to keep changing suppositions about the state of the world.

6 Using Domain Knowledge to Control Search

At any time during planning there are subgoals and normal preconditions which are not yet satisfied and there are operators with variables only partially bound. The planner uses these unsatisfied conditions in its search for viable actions and variable bindings. It may happen that a condition is true in the current data-base state. In this case variable bindings are the values for which the condition holds. Otherwise actions linked to the condition either directly or indirectly are considered. Finding the values for the action variables that allow the action to be legally executed in a current state can be computationally expensive. Clearly, testing every possible combination of variable bindings would be inefficient. In order to control the search for bindings, domain knowledge which is already available in the operator definitions is fully exploited. Syntactic and semantic checks on expressions (preconditions, goals, constraints) are used to prune out nonviable bindings.

Every action that leads to a specified condition through a path mapping operator goals to subgoals or preconditions is a possible first action in satisfying the condition. Once a candidate action is found, its variable bindings are determined as described below.

- Bindings can be found directly from the condition. When the condition is found to be true in the current world state, candidate bindings are picked up directly from the data base. (There can be more than one set of bindings.) For example, a precondition of **build** is created-from(system,baseline). If there is a system in the data base, say *system1*, which has been created from a baseline, *sys1*, a candidate binding for the variable, system, is *system1* and for the variable, baseline, is *sys1*.

- Bindings can be found from operator preconditions which must be true. Since the preconditions of the proposed action must be true, the set or sets of variable bindings can be determined by querying the data base. If the query results in all variables in the precondition being bound, the precondition is true. Again, there may be more than one set of bindings that make the precondition true. New variable bindings can be added to sets and sets of bindings can be discarded as tests are done on action variables propagated up through the path link via mappings. If an operator is linked either directly or indirectly to the action through one of its subgoals, then all of its preconditions must be true. Therefore a proposed set of action bindings can be extended or discarded by querying the data base with such a precondition which must be true. For example the action **make** satisfies the precondition of **build**. In order for the action, **make**, to be legally executed its precondition must be true. If *sys1* is a built system in the data base, then *sys1* is a candidate binding for the variable, baseline.

- Bindings are found from operator constraints which must hold; tests on operator constraints limit the viable alternatives for action bindings. If a set of bindings causes a constraint to be violated, then that set of bindings is discarded.

- Goal clauses are used to limit sets of bindings. If the goal of an operator on the path from an action to a condition is found to be true in the current state (with all of its parameter values bound from a proposed set of action bindings), then the binding set used is discarded. Posting the effects of the action in such a case would serve no purpose.

The above ways of narrowing search for the possible bindings of an action are the planning analogs of narrowing the search for possible interpretations in plan recognition.

Not all the action parameters are necessarily bound while applying the above rules. An action can have a parameter whose value is a new object or an attribute chosen from a predetermined list of attributes. The values for these variables are generated when the effects of the action are simulated.

An action and its bindings are further tested by simulating effects. If the constraints of any operator in the current interpretation are violated by side effects, the set of bindings is thrown out. The violation of a protected, satisfied condition may also become evident when an action is simulated.

Action credibility is determined during the search for bindings. Those operator conditions which are assumptions rather than certainties are also evaluated while testing bindings. The tested bindings of an action can cause a logical expression to be true for certain, true by assumption, false by assumption or false for certain. Looking at the operator definitions; the precondition, applicable(case,system), of **run-test-case** could be false by assumption for a given set of action variable-bindings in which case the credibility of the action taken with these bindings is low.

When no credible (not false for certain) bindings can be found for action parameters, the proposed action is not a viable alternative for satisfying the specified condition.

7 Using Heuristic Knowledge to Control Search

There are two ways that the user can work in planning with the assistant; heuristics influence both. First the user can control the agenda and specify a single unsatisfied goal. The system will return with a proposed action and bindings. Secondly, the user can specify any number of unsatisfied goals and GRAPPLE will prioritize goals and actions. It will choose, using heuristics, the "best" goal to be achieved and the "best" first action to satisfy it.

Ideally the planning system should choose goals and actions that are likely to lead to success quickly. In order to do this the GRAPPLE planning system uses heuristics in incrementally selecting goals and actions from a pool of candidates. When the user specifies more than one goal, only those goals which can be achieved in the current world state are considered as candidates.

The heuristics for choosing a best, first action to satisfy a specified goal are:

- A desirable action should not violate an already satisfied, protected condition and thus impose the necessity to replan.
- A desirable action should be credible and thus be consistent with current assumptions about the state of the world. An action which partially or totally satisfies a goal and is not based on any assumptions is most desirable.
- A desirable action either directly satisfies the condition or reaches it within a small number of path links. A shorter number of path links often corresponds to fewer actions in a plan and few actions means a simple plan which is preferred. What's more, it takes less time to generate a short plan and less time to execute it.

Actions are grouped by the goal that they achieve. When the user specifies more than one goal to be satisfied and wants the planner to achieve the most promising goal first, two heuristics are used. They are:

- Choose the goal which can be achieved by the action with the highest credibility.
- When it is impossible to find a single goal with the first heuristic, choose the goal which has the smallest number of possible credible first actions. This rule is founded on the premise that if there are many ways to satisfy a condition, many possible first actions, then probably options will remain in a slightly altered world state; whereas if there is only one alternative to satisfy a condition it might disappear once another action is taken. In other words it is best to first consider goals that look highly constrained.

The implementation works in the following way. A list of all possible first actions is associated with each unsatisfied condition, specified by the user. Associated with each action is:

- A credibility rating.
- The length of the path from the action to the condition.
- A flag indicating whether or not a previously satisfied, protected condition will be violated by taking the action, making replanning necessary.
- A branching factor. The branching factor is the number of all possible first actions which lead to the specified goal.

The properties of possible actions, listed above, are used by heuristic rules to choose the "best" action associated with each user-specified goal. Actions which do not violate a protected condition and which have the highest credibility are determined. From these, the one or ones with the shortest path are selected. If more than one action meets the above criteria, all are proposed to the user.

When more than one goal is specified by the user, the system generates a "best" action for each goal. It then selects a goal, the one whose associated action has the highest credibility, and if there are ties, the one with the lowest branching factor. The chosen action is proposed and interpreted in the current world state, its effects are posted, and the state of the world is updated.

Obviously in any implementation where execution occurs before planning is completed, it is possible that a commitment can be made to a plan that is less than optimal. This can happen if goals are achieved in their improper order. One way that has been used to solve the problem is to get additional information about goals that allows planning to be delayed [Wilkins, 1988]. An enhancement to our system would be to test the proposed action by making sure that it can eventually lead to the satisfaction of the specified goal. This search might, however, be expensive. Interleaving execution with planning is definitely something one wants to do when the outcome of actions is uncertain which is true for the domains of interest to us. However, there are not yet good solutions to the problem of committing to an inefficient plan. More work can be done in this area.

8 Example

In order to illustrate how the assistant functions an example of a session between a user and the intelligent assistant follows. The world contains operators (see appendix) for building, testing and archiving a system. In addition to these operator definitions there is a set of predicates and default rules, used to evaluate the world state.

The goal of the top-level operator is to build a new system from a baseline system. In the initial world state, given to GRAPPLE, a baseline system,*sys1*, is already built, but no system has yet been created from it. Two test cases which are applicable to *sys1*, a normal case, *normalcase1*, and a base case, *basecase1*, are also part of the initial world.

THE FIRST ACTION

The user chooses not to take the initiative in proposing the first action toward building a system and defers to the assistant. In response GRAPPLE expands the top-level operator, **build**, into preconditions and subgoals and displays them. The user can then specify a condition or a number of conditions. He specifies all three conditions, not knowing where to begin.

****build(toplevel)**

precond1 sys-constructed(notstatic,created-from(system,baseline))

subgoal1 testing-done(tested(system))

subgoal2 status-protected(or(archived(system),archive-waived(system)))

All actions which either satisfy or lead to the specified conditions are generated along with their variable bindings. Syntactic rules narrow the search here. The subgoals can not be considered because the precondition which must be satisfied first does not hold in the current state - no constructed system created from the baseline already exists. The only possible first actions must satisfy the nonstatic precondition sys-constructed of the operator **build**. There are two of them and both make a new system from the baseline. One makes additions to the code and the other makes cosmetic changes (changes in comments or variable names) to the baseline system. The proposed actions are:

make with variable values (system,system1),(baseline,sys1),(ext,big),(type,new), (target,global)

make with variable values (system,system1),(baseline,sys1),(ext,big),(type,cosmetic), (target,global)

Either action is permissible in the current state because the precondition (that the baseline is built) of **make** is satisfied when the variable, baseline, is bound to the object *sys1*. The value of the variable, baseline, is actually found by querying the data base with the precondition of **make**. The value of system, *system1*, is a generated new object and the values of the other variables are chosen from a predetermined list of attributes.

Both actions have high credibility (true for certain) and both directly satisfy precond1. Therefore both actions are equally desirable. The intelligent assistant proposes both actions to the user. The user chooses the first, to make a new system with new changes. The action is interpreted by the recognizer. The only interpretations of the proposed action of interest are the ones in which the action leads to the originally specified condition. The action is interpreted, its effects are posted and the world state is updated. As a result of action effects being posted, assumptions about the world might be altered. For instance when a new system is made from the baseline, the test on the base case that was applicable to the baseline system becomes applicable to the new system by assumption.

SECOND ACTION

The user again wants the assistant to plan for him and he wants an explanation of what has happened; the extended interpretation is displayed.

**** build1(toplevel) status: precond-satisfied**

pre-inst1:sys-constructed status: satisfied and protected

**** make1 status: completed EXPLICIT ACTION**

sub-inst1:testing-done

sub-inst2:status-protected

(*explanation: Operators are starred; the operator listed beneath a condition is the one that achieves it. The expansion of an operator is indicated by indentation.*)

This time the user again wants help in choosing the goal and the action. He specifies both subgoals. All actions which lead to **sub-inst1** and **sub-inst2** are generated. Three primitive actions lead to **sub-inst1**; none of them directly. They are:

study with variable values (system system1), credibility "true for certain", and path length 2.

run-test-case with variable values (system,system1) (case,basecase1) (tr,tr1) (compl,compl) (debug none), credibility "true by assumption", and path length of 3.

run-test-case with variable values (system,system1) (case,normalcase1) (tr,tr2) (compl,compl) (debug,none), credibility "true by assumption", and path length of 3.

The credibility of both actions which run a test case is "true by assumption" because the precondition of each (that the test is applicable to *system1*) is presumed true once the effects of the first action, making a new system from a baseline, are posted and the extended world state is updated . It is an empirical assumption, represented in default rules, that a test case which is applicable to the baseline system is also applicable to a changed system, created from it. As before the variable bindings are picked up from a data-base query with the precondition predicate which must be true. These two actions both have a path length of three. This follows from the observation that the goal of **run-test-case** satisfies the subgoal of the operator **run-cases** which satisfies the subgoal of the operator **test**.

Of the three actions the "best action" associated with **sub-inst1** is **study** because it has the highest credibility, "true for certain", and the shortest path. The branching factor is three, since it is one of three possible first actions which lead to **sub-inst1**.

The subgoal, **sub-inst2**, could be satisfied directly from the current data-base state by changing existing assumptions about the state of the world. The expression for **sub-inst2** has a credibility rating of "false by assumption". Although archiving should not be waived since the changes made were not cosmetic, the TMS predicate, **archive-waived(system)** can be made true by revising the current world assumptions, but it is preferable not to have to do this. The only option for satisfying **sub-inst2** is to change assumptions.

The intelligent assistant determines that **sub-inst1** is the more desirable goal to complete and, using heuristics, proposes the most credible action, **study**. The action is interpreted in the current world state, its effects are posted and the data base is updated.

THIRD ACTION

The user now wishes to take an action on his own and proposes to archive the system. The recognition system can not find a viable interpretation for this action since no testing has been done. The user is so advised. He needs help and uses the assistant's summary of work that has been done.

**** build1(toplevel) status: precond-satisfied**

> **pre-inst1:sys-constructed status: satisfied and protected**
>
> **** make1 status: completed EXPLICIT ACTION**
>
> **sub-inst1: testing-done**

** test1 status: waiting-on-subgoals

sub-inst3:tests-acomplished

sub-inst4:code-studied status: satisfied and protected

** study1 status: completed EXPLICIT ACTION

sub-inst2:status-protected

The user specifies Sub-inst3:tests-accomplished. Sub-inst1 cannot be considered since it is already decomposed into subgoals. There are two possible actions leading to sub-inst3, each with credibility "true by assumption" and with path-length 2. They are:

run-test-case with variable values (system,system1) (case,normalcase1) (tr,tr4) (compl,compl) (debug,none) ,credibility "true by assumption", path length 2, and branching factor 2.

run-test-case with variable values (system,system1) (case,basecase1) (tr,tr3) (compl,compl) (debug none), credibility "true by assumption", and path length of 2, and branching factor 2.

Both are proposed and the user chooses the first. This action is interpreted, its effects are posted and a future world state is generated. It should be noted that if the first action had been to make cosmetic rather than new changes to the baseline system, the base case would be applicable to *system1*, by assumption, but the normal case would not. This is because normal tests are not necessary when only cosmetic changes are make to a system. In this event the credibility of the first action above would be "false by assumption" and the assistant would propose only the second action to the user.

FOURTH ACTION

The user proposes the next action.

run-test-case with variable values (system,system1) (case,basecase1) (tr,tr5) (compl,compl) (debug,none).

It is accepted and the extended interpretation becomes:

** build1(toplevel) status: waiting-on-subgoals

pre-inst1:sys-constructed status: satisfied

** make1 status: completed EXPLICIT ACTION

sub-inst1:testing-done status: satisfied and protected

** test1 status: completed

sub-inst3:tests-acomplished status: satisfied

** run-cases1(iterated) status: completed

sub-inst5:single-case-run status:satisfied

** run-test-case1 status: completed EXPLICIT ACTION

** run-cases2(iterated) status: completed

sub-inst6:single-case-run status: satisfied

** run-test-case2 status: completed EXPLICIT ACTION

sub-inst4:code-studied status: satisfied

** study1 status: completed EXPLICIT ACTION

sub-inst2:status-protected

FIFTH ACTION

There is one subgoal, sub-inst2, remaining to be satisfied. The user feels confident enough to propose archiving the system. Because the system has now been tested it can be archived. With this action the top-level goal is satisfied and the plan is completed.

** build1(toplevel) status: completed
    pre-inst1:sys-constructed status: satisfied
    ** make1 status: completed EXPLICIT ACTION
    sub-inst1:testing-done status: satisfied
    ** test1 status: completed
        sub-inst3:tests-acomplished status: satisfied
        ** run-cases1(iterated) status completed
            sub-inst5:single-case-run status: satisfied
            ** run-test-case1 status: completed EXPLICIT ACTION
        ** run-cases2(iterated) status: completed
            sub-inst6:single-case-run status: satisfied
            ** run-test-case2 status: completed EXPLICIT ACTION
        sub-inst4: code-studied status: satisfied
        ** study1 status: completed EXPLICIT ACTION
    sub-inst2:status-protected status: satisfied
    ** archive1 status: completed EXPLICIT ACTION

Credibility was the determining heuristic used in choosing actions in the above example. In another example where the goal is to construct a new system, both the value of the action's branching factor and the length of an action's path guide the generation of a correct plan. In the interest of clarity the example is simple. However, the techniques described here are applicable to larger sets of operators and more complex worlds. In a more complicated situation the advice of an automated intelligent assistant might be truly beneficial to the user. GRAPPLE does depend on domain knowledge being represented in a set of hierarchical operators. In a complex world determining the operators and default rules is difficult and requires expertise.

9 Conclusions

We have described an intelligent, domain independent assistant which works cooperatively with a user, recognizing his plan and planning for him. An advantage of this system is that planning can occur anytime help is desired. The plan hierarchy controls the development of an ongoing interpretation of actions and thus directs search. Knowledge embedded in operator definitions is continually checked against the current world state in order to bind variables to values that satisfy constraints. Empirical knowledge allows the planner to determine credible actions and the recognizer to determine credible interpretations. Heuristic rules allow the assistant to propose the best choices of new actions based on credibility, path length, and branching factors. The intended effect of the heuristics is to keep the plan simple and credible, to avoid mistakes and dead ends, and consequently to avoid the necessity of replanning. By integrating various kinds of knowledge, search can be controlled. The result is that the user and the assistant can work together interactively and effectively accomplish goals.

Appendix
Example Software Development Operators

build

```
operator( build(system,baseline),
goal( built(system) ),
preconds( sys-constructed(notstatic, created-from(system,baseline)) ),
subgoals( testing-done(tested(system)),
        status-protected(
                or(archived(system), archive-waived(system)) ) ),
effects( add(built(system))).
```

make

```
operator( make(system,baseline,ext,type,target),
goal( created-from(system,baseline) ),
preconds( baseline-built(static, built(baseline)) ),
effects( new(system,system),
                add(created-from(system,baseline)),
                add(percent-change(system,ext)),
                add(type-change(system,type)),
                add(target-of-change(system,target)) ) ).
```

archive

```
operator( archive(system),
goal( archived(system) ),
preconds( testing-done(static, tested(system))),
effects( add(archived(system)) ) ).
```

test

```
operator( test(system),
goal( tested(system) ),
subgoals( tests-acomplished(cases-run(system)),
        code-studied(studied(system)) ),
effects( add(tested(system)) ) ).
```

run-cases

```
operator( run-cases(system,case,tr),
keywords( iterated-until(testing-done(system)) ),
goal( cases-run(system) ),
subgoals( single-case-run(testevent(system,case,tr))),
effects( add(cases-run(system)) ) ).
```

run-test-case

```
operator( run-test-case(system,case,tr,compl,debug),
goal( testevent(system,case,tr) ),
preconds( case-is-appropo(static, applicable(case,system)) ),
effects( new(testresult,tr),
                add(testevent(system,case,tr)),
                add(case-on-system(case,system)),
                add(completion(tr,compl)),
                add(debug-usage(tr,debug)) ) ).
```

study

```
operator( study(system,view),
goal( studied(system) ),
effects( add(studied(system)) ) ). .
```

In addition to these operator definitions there is a set of predicates and default rules, used to evaluate assumptions about the world state. They are not shown here.

References

[Connell,Huff,Lesser, 1989] Connell, M.E., Huff, K.E., and Lesser, V.R., *Implementing an Incremental Hierarchical Plan Recognition System* COINS Technical Report 89-82: University of Massachusetts, Amherst, MA. September 1989.

[Croft, 1988] Croft,W.B.and Lefkowitz,L.S. "Using a Planner to Support Office Work" in *Proceedings of the ACM Conference on Office Information Systems*, March, 1988.

[Doyle, 1980] Doyle, J. "A Truth Maintenance System", *Artificial Intelligence*, 12(1980),231-272.

[Genesereth, 1979] Genesereth, M.R. "The Role of Plans in Automated Consultation." in *Proceedings IJCAI-79*, Palo Alto, CA: Morgan Kaufmann, 1979,311-319.

[Huff, 1988a] Huff, K.E. and 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, 1987] Huff, K.E. and Lesser, V.R. *The GRAPPLE Plan Formalism* COINS Technical Report 87-08: University of Massachusetts, Amherst, MA. 1987.

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

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

[Miller and Goldstein 1977] Miller, M.L. and Goldstein, I.P. "Structured Planning and Debugging" *Proceedings IJCAI-77*

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

[Sacerdoti,1979] Sacerdoti, E. D. "Problem Solving Tactics", *Proceedings of IJCAI-79*, Tokyo, Japan,pp. 1077-1085, 1979.

[Swartout, W.] Swartout, W., editor, "Santa Cruz Workshop on Planning", *AI Magazine* vol. 9 no. 2, (1988), pp. 115-130.

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

[Wilkins, 1988] Wilkins, D.E. *Practical Planning.* San Mateo, CA: Morgan Kaufmann,1988.