Plan-Based Intelligent Assistance: An Approach to Supporting the Software Development Process

Karen Erickson Huff
Ph.D. Thesis
Computer and Information Science Department
University of Massachusetts

COINS Technical Report 89-97

PLAN-BASED INTELLIGENT ASSISTANCE:

AN APPROACH TO SUPPORTING

THE SOFTWARE DEVELOPMENT PROCESS

A Dissertation Presented

by

KAREN ERICKSON HUFF

Submitted to the Graduate School of the
University of Massachusetts in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

September 1989

Department of Computer and Information Science

PLAN-BASED INTELLIGENT ASSISTANCE: AN APPROACH TO SUPPORTING THE SOFTWARE DEVELOPMENT PROCESS

A Dissertation Presented

by

KAREN ERICKSON HUFF

Approved as to style and content by:

Victor R. Lesser, Chairperson of Committee

W. Bruce Croft, Member

Jack C. Wileden, Member

Charles Rich, Member

W. Richards Adrion, Department Head
Computer and Information Science Department

ACKNOWLEDGMENTS

I would like to thank my advisor Victor Lesser for his guidance, support, and encouragement during the course of this research. As a relative latecomer to the serious study of artificial intelligence, I wanted to learn not only the techniques but the "spirit" of the field. Victor's approach to identifying the central intellectual issues, framing the key questions, and distinguishing between dead-end paths and promising avenues of exploration served as my model of how AI research is conducted. His perspective helped to sharpen and enhance many aspects of the work reported in this dissertation.

ABSTRACT

PLAN-BASED INTELLIGENT ASSISTANCE: AN APPROACH TO SUPPORTING THE SOFTWARE DEVELOPMENT PROCESS

SEPTEMBER 1989

KAREN ERICKSON HUFF, B.A., OBERLIN COLLEGE
M.S., STANFORD UNIVERSITY
Ph.D., UNIVERSITY OF MASSACHUSETTS

Directed by: Professor Victor R. Lesser

We describe an approach to *intelligent assistance* for the *process* of software development, using an architecture based on *planning* and *plan recognition*. Processes are formally defined hierarchically via plan operators. Plans are constructed dynamically from the operators; the sequences of actions in plans are tailored to the context of their use, and conflicts among actions are prevented. Monitoring of the development process, to detect and avert process errors, is accomplished by plan recognition; this establishes a context in which programmer-selected goals can be automated via plan generation. Achieving substantive assistance requires extensive domain knowledge. We show two ways to extend the representational power of planning systems to capture additional domain knowledge.

Intelligent assistance involves, by its very nature, *hidden state*—information about the state of the world is incomplete. This makes it impossible to evaluate some preconditions, subgoals or constraints in operator definitions. The solution to acquiring the missing information is to make plausible assumptions using knowledge about what is typical or

expected in the current context. This empirical knowledge allows additional domain knowledge to be exploited, yielding a deeper context for reasoning about actions.

Plan recognition can be extended to handle hidden state by representing additional domain knowledge nonmonotonically in a truth-maintenance system. *Credibility*, the degree of agreement with current assumptions about the world, provides the basis for distinguishing alternative plans. A plan that violates some current assumptions can still be pursued after revising those assumptions. We define *reconciliation*, a new approach to revising assumptions that both propagates the consequences of adopting the correct assumptions as well as providing a rationale for those assumptions.

Some knowledge about the domain directly concerns the modification of plans. This knowledge lends itself to a transformational representation that allows for operations like substituting one goal for another or adding a new subgoal. Since these transformations act upon plans as the state of the world, they can be formalized as *metaoperators* and synthesized into *metaplans*. This not only provides a single formalism in lieu of adding many special purpose constructs to the operator definition language, but also extends the role of metaplanning in planning systems.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1
PLAN-BASED INTELLIGENT ASSISTANCE

1.1 Towards a New Form of User Assistance

Computer-based work environments, such as CAD/CAM workstations, automated offices, and software development environments, are growing increasingly complex, and increasingly difficult to use. New forms of assistance are needed so that users can effectively employ the powerful features these environments now provide.

Existing methods of assisting users—including power tools, programmable command languages, HELP systems, and user interfaces—provide support from a *local* perspective, without regard for context formed by preceding or future actions. Power tools allow complex activities to be carried out at a higher level of abstraction, relieving the user of many tedious details. Programmable command languages let users tailor their environments by automating commonly used sequences of closely related commands. HELP systems, usually geared to novice or intermediate level users, answer questions about specific commands and their parameters. User interfaces based on natural language or graphics are concerned with more effective means of communication between the user and the environment, not with the level or content of the communication.

1.1.1 Support for Processes

To augment these means of assistance, there is a place for assistance based on a *global* perspective towards user activities. Activities do not occur in random sequences, but follow recognizable *processes* (in software development and CAD/CAM) or *procedures* (in the office). These processes are hierarchical, although only the lower levels of the hierarchy are manifest in the commands provided in current environments. Some activities are related: one may be part of another, or a precondition to it; some activities are not related at all: they contribute to entirely separate tasks. Typically, users will have multiple tasks in progress concurrently.

Software development environments, for example, have traditionally lacked a process viewpoint. Separate parts of the software development process are typically supported by separate tools, while global patterns of tool usage are not made explicit, and are not exploited. Thus, the environment cannot prevent a programmer from starting compilations before an appropriate context is set up, enforce a policy of regression and performance testing before a customer release, insure that new source versions are checked back into the source code control system, or guarantee that source files are deleted only after their contents have been archived or superseded. This support, only achievable by taking a process viewpoint, would be valuable to both programmers and their managers.

To the extent that processes are supported at all in existing environments, the support can be characterized as highly localized automation. Command language scripts and special tools such as MAKE [Feldman, 1977] are the means to this automation. In this picture, the programmer provides the global framework in which local activities (some automated, some not) occur. The programmer keeps track of what has been done and what still needs doing. The programmer, not the system, knows why particular tools are invoked, when they should be invoked, and when they should not be. When exceptional situations arise,

it is up to the programmer both to recognize that a standard script cannot be used, and to generate and carry out an appropriate variation on the standard solution. Only the programmer knows what interrelationships exist among the separate actions being performed.

Obviously, given this existing situation, there are many opportunities for getting the machine into the process loop. One approach to providing process support, which is evolutionary in spirit, involves increasing the existing level of automation. In this case, there is no attempt at comprehensive coverage of processes; rather, there is explicit representation of only those aspects of processes that can be automated. This approach is exemplified in work on *active environments* (a term introduced in [Balzer, 1987]). An active environment is one that recognizes opportunities for taking actions without having to be explicitly told, via specific commands, to take those actions; examples of active environments are GENESIS [Ramamoorthy et al., 1985], CLF [Balzer, 1987], and Marvel [Kaiser & Feiler, 1987].

1.1.2 Intelligent Assistance

An alternative approach, and the one taken here, is to make comprehensive representation of processes the primary objective, letting this representation determine the nature of the support to be provided. By comprehensive, we mean that the representation should capture knowledge about all parts of a process; knowledge about the parts that cannot be automated will be incomplete, which is quite different from being missing entirely. This focuses attention on the fact that automation is not the only way to assist a programmer carrying out a process. Three examples of other types of assistance are generating agendas of tasks to do, summarizing work accomplished, and giving advice about proposed courses of action. Thus, a representation should be judged on its ability to

capture knowledge of the global framework for actions, previously held only by the programmer, on which both automation and other types of assistance can be based.

In this view, process support will take the form of a partnership between the user and an *intelligent assistant.* The intelligent assistant will be a visible agent that can be directed by the programmer, an "apprentice" in the spirit of [Rich & Shrobe, 1978]. In contrast, active environments provide support invisibly; they do not require dynamic control from the programmer—in fact, they do not allow it. In a partnership, the user will have to perform certain activities (for example, editing source code) and make certain decisions (which modules need changing, or what archiving method to use); the remaining activities can be performed by *either* partner.

An effective partnership requires that the intelligent assistant be based on a system for reasoning about the hierarchical structure, purpose, and goals of actions. Only if the assistant understands the structure and rationale for actions can supporting tasks be automated in an appropriate manner—through a "plan" that takes advantage of what the user has already accomplished without interfering with other ongoing activities. Only if the assistant can analyze the purpose of the user's actions can advice be volunteered when these actions do not appear to be consistent with previous actions and known goals. An explicit representation of the interrelationships of actions can also be used to summarize past accomplishments, explain the purpose of actions already taken, generate an agenda of tasks yet to be performed, and offer plans for carrying out those tasks.

In the next section, we introduce a plan-based approach to achieving this type of intelligent assistance, using the software development environment as the example domain [Huff & Lesser, 1988].

1.2 A Plan-based Architecture

One system for reasoning about actions is *planning*, an AI approach to a theory of actions [Genesereth & Nilsson, 1987]. Under this paradigm, a process is viewed as a mapping between domain *goals* and environment *actions* (Figure 1.1). Typical software development goals are concerned with adding functionality to a system version, fixing bugs, adhering to various project-specific policies, and maintaining an organized on-line workspace. The actions available to achieve these goals consist of invocations of tools provided within the environment, such as editors, compilers, and analysis tools. Knowledge of a specific software process governs the mapping from goals to actions, distinguishing between appropriate sequences of actions and random ones.

In planning (Figure 1.2), knowledge of a domain is expressed in *operators* (parameterized templates defining the possible actions of the domain) together with a *state schema* (a set of predicates that describe the state of the world for that domain). *Goals* are logical expressions composed of the state predicates. A *plan* is a hierarchical, partial order of operators (with bound parameters) that achieves a goal given an initial state of the world.

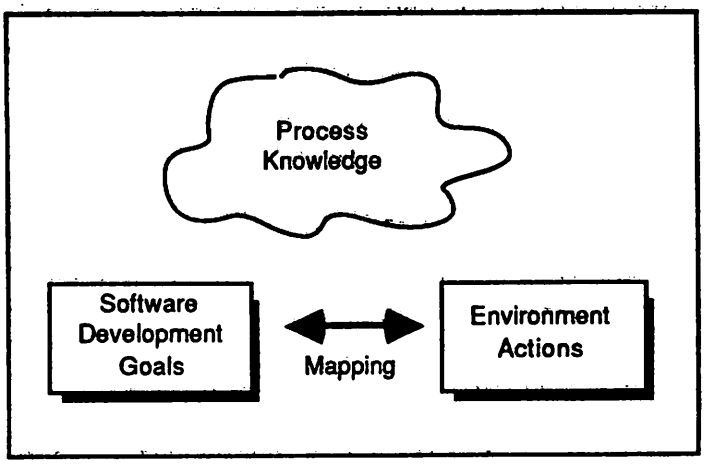

Figure 1.1 A View of Software Processes

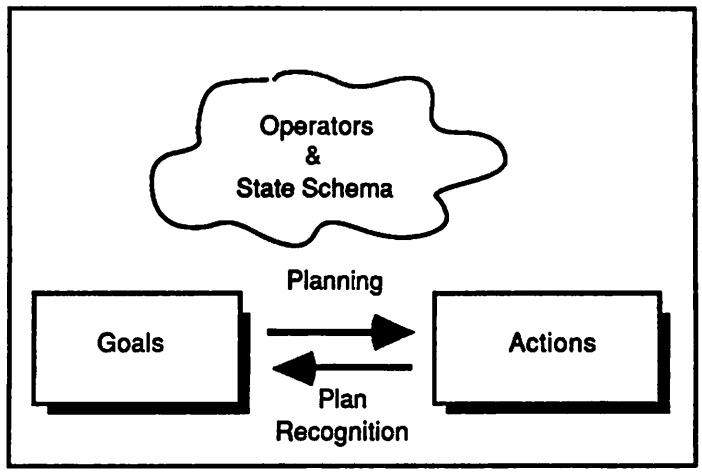

Figure 1.2 The Planning Paradigm

There are two mapping algorithms: *planning*, where a plan is constructed given a goal and an initial state, and *plan recognition* [Allen, 1983; Azarewicz et al., 1986; DeJong & Mooney, 1986; Genesereth, 1979; Kautz & Allen, 1986; Sidner & Israel, 1981; Wilensky, 1981], where a plan and its goal are inferred given a sequence of actions and an initial state.

When the planning paradigm is applied to software processes, operators are the vehicle for formally defining processes; plans are the data structures that represent instantiations of processes; and, intelligent assistance is based on the algorithms of planning and plan recognition. If the programmer retains the initiative for performing the process, issuing commands exactly as at present, plan recognition can detect and avert process errors. This "kibitzing" is an automated version of a colleague watching over the shoulder of a programmer at work. Alternatively, the programmer can state a goal to be satisfied, invoking plan generation to automate achievement of the goal. Depending on the scope of the goal, plan generation may be completely automatic, or cooperative [Croft & Lefkowitz, 1988; Lefkowitz & Croft, 1989] (relying on interactive input from the programmer).

This form of intelligent assistance, combining volunteered advice and cooperative automation, extends "machine mediation" of software development (a term used in [Balzer et al., 1983]) to cover the development *process* as opposed to the developing *products*. The product orientation of plan-based intelligent assistance is evident in two systems directed at understanding the deep structure of code [Johnson & Soloway, 1985; Rich & Waters, 1988].

1.2.1 Software Process Operators

As an example of a software process, consider how implementation might be carried out for a traditional programming language such as C, assuming currently accepted engineering practice such as incremental development, source code control, bug report database, and specialized test suites. A partial library of operators for this domain appears in Figure 1.3. A state schema supporting these operators is sketched in Figure 1.4, using the ER model of data [Chen, 1976] as the graphical presentation; relationships and attributes correspond to the logical predicates used in the operator definitions.

These operator definitions follow the state-based, hierarchical planning approach [Sacerdoti, 1977; Tate, 1976; Wilkins, 1984]. Each operator has a *precondition* defining the state that must hold in order for the action to be legal, and a set of *effects* that defines the state changes that result from performing the action. These core clauses are augmented by a *goal* clause that defines the principal effects of an action (thus distinguishing them from "side-effects" of the action), and a *constraints* clause that defines restrictions on parameter values. We give an example to show how these clauses are used.

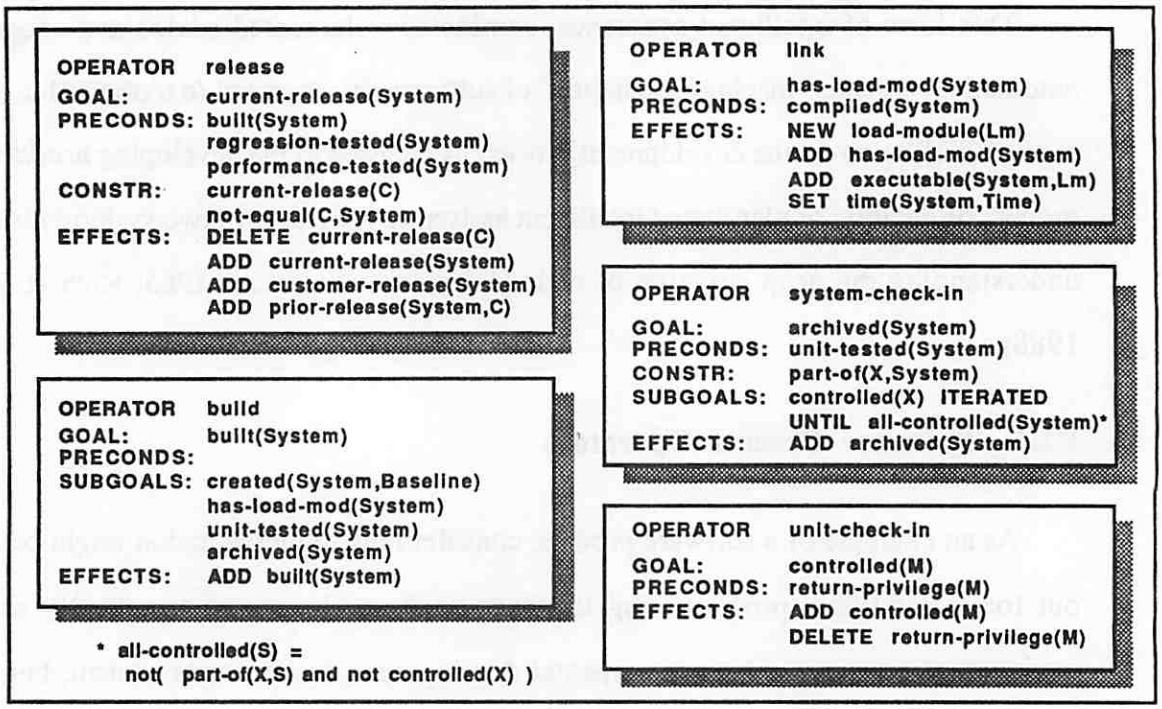

Figure 1.3 Software Process Operators

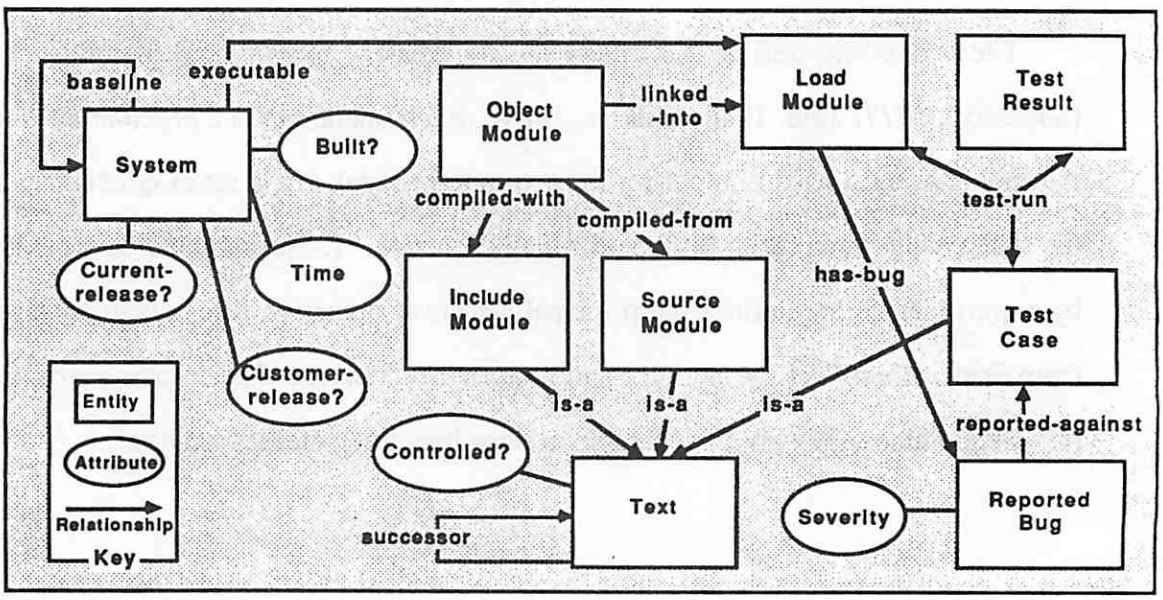

Figure 1.4 Software Process State Schema

The *unit-check-in* operator in Figure 1.3 describes the action of checking a new version of a source module into a source code control system* (such as RCS [Tichy, 1985]). The precondition on the action requires that the module was previously checked-out with return privileges. The goal of the action is that the new version is now "under" source code control; this is also one of the effects of the action. There is one side-effect, namely that the return privilege is lost (another operator is used to describe the type of check-in that retains the return privilege).

An abstraction hierarchy is created through *complex* operators having *subgoals*. (The use of abstraction is essential for describing complicated processes.) The subgoals of the *build* operator decompose building into four parts: creation of new source versions, creation of a load module, running unit tests, and checking-in. *Primitive* operators, which do not have subgoals, correspond to the atomic actions in the domain. Some primitive actions, like *unit-check-in*, correspond to tool invocations (perhaps with specific parameter settings). Other primitive actions correspond to command language scripts; *release* could be implemented as a script that copies a load module to the customer's directory and issues a release notice.

---

* A source code control system provides three functions useful in software development. The first is compact storage of historical versions, achieved by storing differences between them rather than storing each version as is. The second function is to provide a naming convention that captures the relationships among the versions; thus version 1.1.1 follows version 1.1 and shares the same antecedent with version 1.1.2. The third function is a system of "keys" that prevents two programmers from modifying the same version at the same time. If programmer A has the key to version 1.1, then he has the right to create a new version that is a successor to version 1.1; while he has the key, other programmers are able to look at version 1.1, but not to place a modified version of it back in the system. Programmers *check-out* module versions from the system (with or without requesting keys, e.g., the return privileges) to get copies to work on and *check-in* new versions (assuming they have the keys) to save versions for possible later use.

All the policies and procedures that are associated with a particular software development process must be included in the operator definitions. Changing the definitions of the operators changes the process that will be followed. For example, the *release* operator in Figure 1.3 requires that performance tests and regression tests be run before a customer release is made. These requirements could be relaxed (by deleting one or both testing-related preconditions) or strengthened (by adding another precondition requiring execution of a particular analysis tool or updating of release documentation).

The operator definitions can be used to define processes that are as structured or as flexible as desired. Specific ordering of activities can always be enforced through preconditions. Another example relates to the *build* operator of Figure 1.3, which includes the requirement for *archiving* the system. In a highly structured process, archiving might be considered achieved only if the source modules were checked into the source code control system. In this case, only one operator is defined that has *archived(System)* as its goal. In a less structured process, either source code control or local backup (simply duplicating a file) might be considered appropriate. In this case, there would be two competing operators that have *archived(System)* as their goal. In an even more flexible process definition, the archiving subgoal of *build* might be dropped, and archiving treated as a top level goal in its own right (thus, not formally connected to other activities).

1.2.2 Software Process Plans

Plans are constructed dynamically by instantiating operators. Operator definitions contain sufficient information to reason about sequences of actions without actually executing the actions. The state changes that an action causes are explicit; therefore, sequences of actions can be "simulated" and future states of the world "projected". The exact preconditions for an action are explicit; therefore, actions can be selected and ordered correctly. Plans are automatically tailored to the exact context for which they are needed. If

a subgoal has already been achieved as a side-effect of prior activity, actions to achieve that subgoal will be omitted. If part of a plan fails during execution, replanning will fill the gaps left by the failure (and only those gaps).

A partial example of a hierarchical plan is given in Figure 1.5. There, a vertical slice covering three hierarchical levels is shown; lower levels reveal more detail in the plan. Downward arrows between levels connect desired states with operators instantiated to achieve them; in general, there may be several alternative operators that could achieve a given state. Arrows within levels show how the achievement of certain states is partially

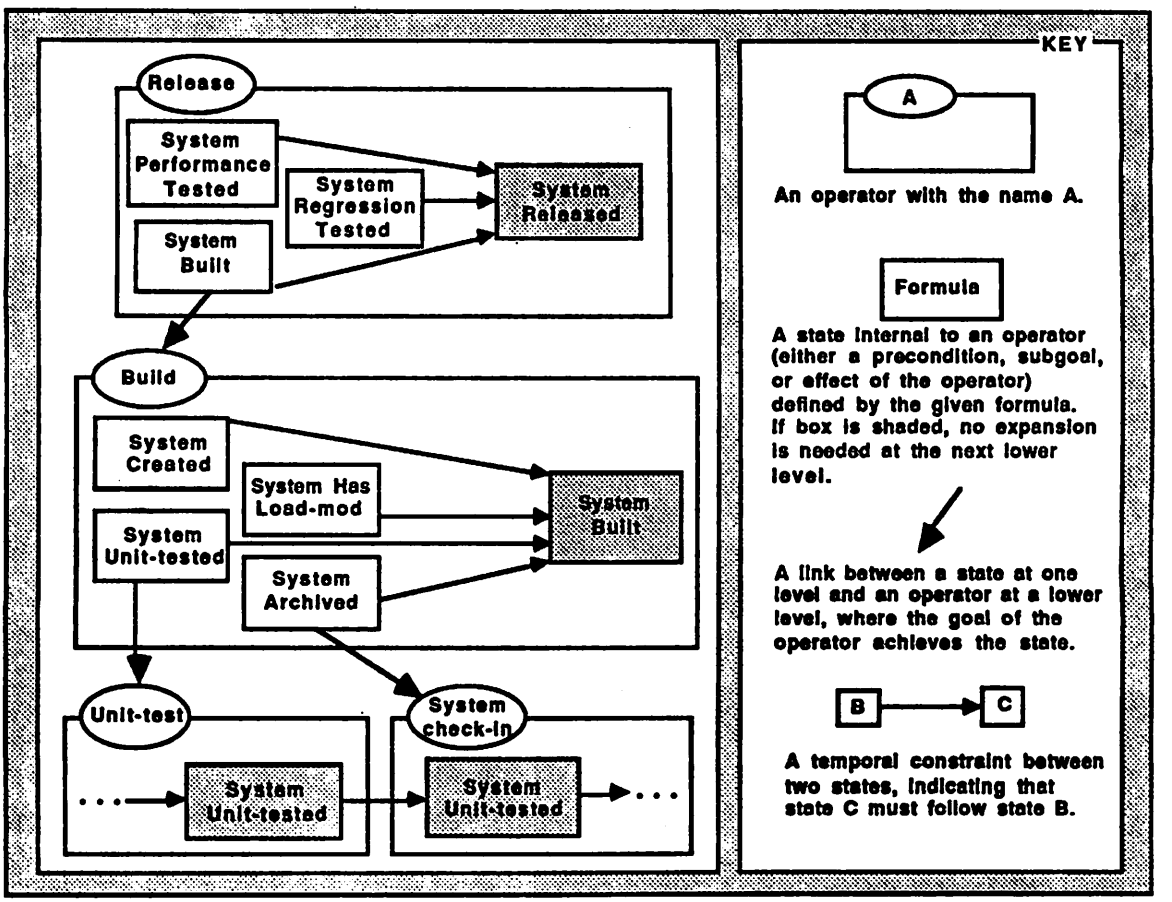

Figure 1.5 A Vertical Slice through a Hierarchical Plan

ordered with respect to time. These temporal constraints follow from the operator definitions: precondition states must always precede subgoals, for example.

Interleaving of actions from different plans or subplans is implicitly allowed, subject only to the stated preconditions. For example, the subgoals of *build* can proceed in parallel, subject to the preconditions of the operators chosen to satisfy them. We would expect these operators to be written in such a way that preparation of test cases (part of *unit-testing*) could be intermixed with editing of source modules (part of creating the system components from the baseline system). However, actually executing the test cases obviously cannot be done until the load module has been created. Interleaving is not limited to actions within the same plan; programmers may have several plans in progress simultaneously, and will typically intermix actions from different plans.

1.2.3 Integration in an Environment

The integration of an intelligent assistant into an environment is shown in Figure 1.6. The base environment contains a command language processor that invokes the available

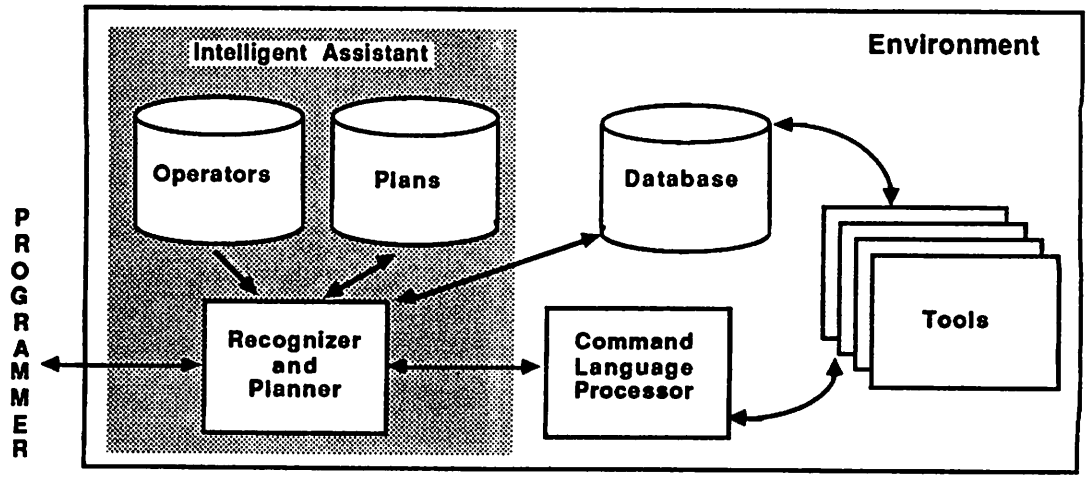

Figure 1.6 Intelligent Assistant Integrated with Environment

tools. The tools in turn reference and update the environment storage system, implemented as a database [Bernstein, 1987; Huff, 1981; Penedo & Stuckle, 1985] or objectbase [Wile & Allard, 1986], comprising not only software products but also their attributes and the relationships among them. The intelligent assistant has an active component based on algorithms for plan recognition and plan generation. These algorithms are process independent; they obtain process knowledge by referencing a library of operators. In order to construct plans, the algorithms must reference the current state of the world, which is essentially the database already present in the environment.

The introduction of an intelligent assistant expands the role of the database in the environment. Information that is not actually recorded in the database but potentially available from tools may be worthwhile capturing because it is useful to the intelligent assistant. And, the intelligent assistant may produce new information that it needs itself, and that the tools may find useful. As an example of additional state information, consider the release operator. The effects of *release* define the "meaning" of release from a higher-level perspective than the script implementing release. The script invokes the *copy* command to copy the contents of one file to another; *copy* is not aware that a release is being copied or that the release sequence is being extended. This higher-level view is needed by the intelligent assistant, but could be used by tools to tailor their activities to the larger context.

The programmer communicates with the intelligent assistant, which moves smoothly between passive and active assistance. For commands issued in the normal way, plan recognition is invoked to find an interpretation for the action. If a valid interpretation is found, the command is passed to the command language processor for execution. An *interpretation* is a path from the action up through the plan hierarchy to a top level operator of an existing or new plan; an interpretation is *valid* if all relevant preconditions and constraints on the path are satisfied. When the programmer issues the command *achieve*

<*goal*>, plan generation is invoked to produce a sequence of commands for execution by the command language processor. Plan generation can also be invoked to satisfy a precondition to make an interpretation valid. Specific examples are presented in the next section.

1.2.4 Examples of Intelligent Assistance

Consider the plan recognition situation diagrammed in Figure 1.7. Here, the programmer has been building a new system version (S1), and work has proceeded as far

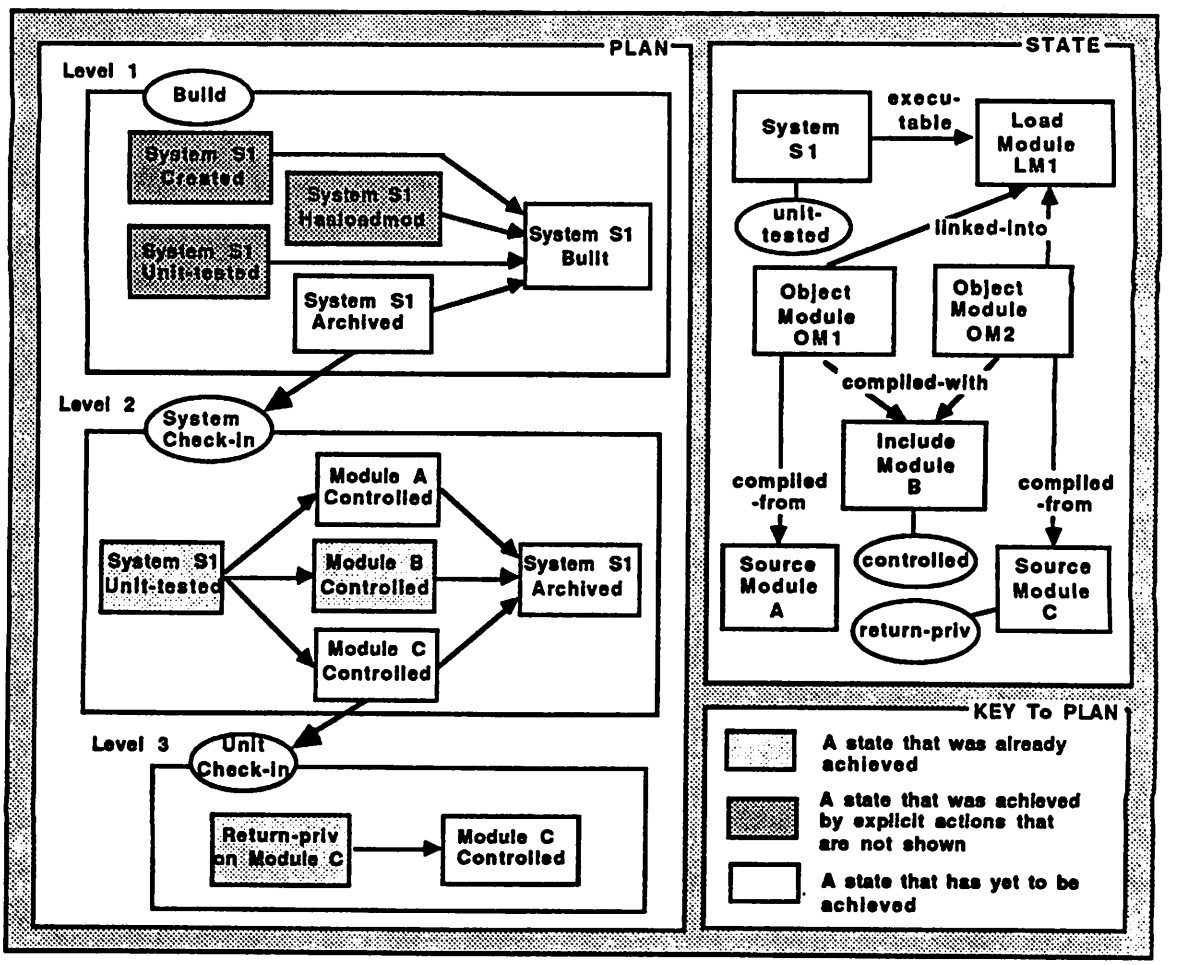

Figure 1.7 A Plan in Progress

as unit-testing. Thus, three subgoals of *build* (at level 1) have been satisfied (the detail of exactly how this was accomplished has been omitted). The current state of the world is also diagrammed, showing that S1 was constructed from two source modules and one include module. Assume that the programmer has just issued a command for *unit-check-in* on module C. This action is "recognized" as fitting into the plan for building S1, because *unit-check-in* (on level 3) satisfies a subgoal in *system-check-in* (on level 2) which satisfies the remaining subgoal in *build* (on level 1). This interpretation for *unit-check-in* of C is valid, because all necessary preconditions are met: return privileges exist for C (level 3) and S1 has been unit-tested (level 2). With a valid interpretation, no advice need be volunteered.

Since the roles and interrelationships of actions are explicitly represented in plans, agendas (what needs doing) and status reports (what has been done) can be generated at multiple levels of abstraction. A plan can be constructed for any agenda item, and the rationale provided for any completed activity. In contrast, the UNIX™ history mechanism keeps all previous commands (up to some pre-specified limit) in a text file; old commands can be retrieved, edited and executed, but the knowledge to do this resides with the programmer, not within UNIX. The DSEE task management facilities [Leblang & Chase, 1984] allow users to associate actions performed with a task/subtask structure; as with UNIX, any intelligent processing of stored task information must be provided by the programmer.

Three additional examples of plan recognition and the support that can be provided from that perspective are:

(1) If no return privileges exist for C, then the interpretation of *unit-check-in* of C is not valid due to an unsatisfied precondition at level 3. Planning

---

Unix is a trademark of AT&T Bell Laboratories.

can be invoked to make the interpretation valid. A plan to do a check-out (to acquire the return privilege) would be generated and executed.

(2) If the programmer had issued a command for *unit-check-in* on module B, this action would be recognized as superfluous—its goal is already true (presumably, B was not modified to make S1).

(3) If the programmer had attempted to relinquish the return privileges on C (perhaps meaning to do a check-in but garbling the parameters on the command), the action could be "doubly" an error. Return privileges on C are necessary to doing *unit-check-in* of C, which is necessary to completing the build of S1. Since the proposed action interferes with other actions, the programmer is asked to confirm the action before it is executed (assuming the interpretation is otherwise valid). However, there may be no valid interpretation for this action (e.g., no "reason" to do a relinquish); then an error would be reported.

As an example of how planning and plan recognition are complementary, consider the request *achieve built(S1)*. The goal is that of the partially completed *build* plan, so the request equates to completing that plan. If this happens after unit-testing is complete, there will be one remaining subgoal in *build* to satisfy. The *system-check-in* operator (level 2) will be chosen to satisfy it, and *unit-check-in* (level 3) will be chosen to expand two subgoals in *system-check-in*; since B is already checked-in, the third subgoal is vacuously satisfied. Because the programmer does not have return privileges for A, a further expansion of the precondition in *unit-check-in* for A will have to be done. The final plan consists of three actions: two *unit-check-ins*, one of which is preceded by a *unit-check-out* to acquire the return privilege.

Planning can be invoked on any level goal within an existing plan, or on new goals. Plan recognition sets the context for planning in three ways. First, it provides additional state information not otherwise available; for example, we have already mentioned that the *release* operator recognizes the additional significance that is inherent in copying a load module to a customer's directory. Secondly, when the goal is part of a recognized plan in progress, some planning choices, such as parameter bindings, may already be decided. Thirdly, by allowing existing plans in progress to be made explicit, plan recognition provides constraints on the plans by which new goals can be achieved. This is discussed further in section 1.3.3.2.

1.3 Key Aspects of a Plan-based Approach

There are three key aspects of the plan-based approach just described: domain-independence, mixed initiative support at multiple levels of abstraction, and explicit representation of goals. These features are discussed below.

1.3.1 Domain-independence

The planning approach is, by design, domain-independent—a distinct advantage in the intelligent assistant application. Domain knowledge is encapsulated in the operator definitions, which are treated as data by the general-purpose algorithms for plan generation and recognition. The plan-based architecture for intelligent assistance can therefore be applied to the other computer-based domains (CAD/CAM, automated office) merely by supplying a domain-specific set of operators. Examples for the office application appear in [Croft & Lefkowitz, 1984]. Furthermore, within a given domain, the actual process supported can easily be changed by modifying the operator definitions. Software processes differ from project to project, just as office procedures vary from office to office.

1.3.2 Mixed Initiative Support at Multiple Levels of Abstraction

The integration of plan recognition and planning allows mixed initiative support, which promotes responsiveness to user needs. The programmer decides when to take control of the initiative and when to hand it off to the intelligent assistant. (Note that operators are defined uniformly, without any indication of whether they will be executed by the programmer or the intelligent assistant.) In a system that provides automation (e.g., planning) only, the initiative always resides with the intelligent assistant; in that case there is no option for the programmer to perform directly a sequence of activities to meet a goal. For goals that can only be automated after significant programmer input, the programmer will be badgered into answering all the questions instead of just being able to do the necessary work without hindrance. In a mixed initiative system, the programmer has the choice of leading or being led; for a given task, the choice may be resolved differently on different occasions—a programmer might choose to be led passively through one task if he is mentally preoccupied with the intellectual challenges of another task in progress.

In addition to being able to decide where the initiative lies, the programmer can also control the level of abstraction at which the interaction will take place. This is possible because activities, and goals, are represented at multiple levels, showing different degrees of detail. Thus, the programmer can choose to interact at a high level, by asking the intelligent assistant to automate a high level goal or by asking for the status of a high level goal. Alternatively, the programmer can choose to interact at a low level, by naming a lower level goal.

1.3.3 Explicit Representation of Goals

Planning is distinguished from other theories of actions by its emphasis on goals to achieve, not actions to perform. Determination of actions proceeds from goals, so that contingency handling (e.g., when some subgoals or preconditions have already been

achieved, or when operators fail) is internal—not external—to the planning system. When process definition is procedural [Osterweil, 1987] or event-based [Huff & Lesser, 1982], the composition and ordering of actions is predetermined; any contingency handling must be built into the definition by hand, in advance. A behavioral approach to modeling software processes using an event description language similar to [Huff & Lesser, 1982] is described in [Williams, 1988].

1.3.3.1 Plans versus Scripts

The implications of making goals explicit, and determining actions from goals, are best seen by contrasting planning with a procedural approach.

Programmable command language scripts (for example, UNIX shell scripts) are essentially procedural descriptions of small portions of software processes. But scripts do not scale up well to represent larger chunks of processes because they lack the information necessary to dynamically adjust themselves to the current context. Scripts are descriptions of actions to be executed—all the reasoning about the structure and rationale of the actions has already been performed, and only the net result of that reasoning is reflected in the script. Since the knowledge about goals, preconditions and effects of actions is missing, the reasoning behind the script cannot be reconstructed; therefore, there is no basis for modifying the script in any way.

Consider a script that accomplishes activity A by doing B followed by C. There is no information in the script that indicates what it is about B (if anything) that is required before C can be done, or what contribution B and C make respectively to achieving the goal of A. Because this information is lacking, there is no way to opportunistically take advantage of B having already been achieved when A starts, or, in the case of a partial failure of B, to tell if C can still be performed, or, in the case of a partial failure of C, if A has still completed successfully. Elaborating the script with conditionals (so that, for example, B is

only executed if its goal is not already achieved) only provides tailoring to exactly those special cases anticipated by the writer. This makes the scripts harder to write, without any compensating gains in the generality of the underlying formalism.

Scripts may also have hidden assumptions about when they are applicable, or what other potentially concurrent activities may interfere with the script. One hidden assumption has to do with continuous execution. Since scripts are currently used for very localized activities, they always execute to the finish once started. However, if scripts are used for activities over a larger scope, then continuous execution is not always possible. Programmers generally have multiple tasks in progress simultaneously, and need to interleave the work on these tasks. Once a script is started, and then interrupted, there is no way to "protect" what the script has already accomplished, so that when the script is re-started, it can continue from where it left off. The ability to do just this is built into planning systems in the form of conflict handling, as described below.

1.3.3.2 Handling Conflicts

A special type of contingency arises when there are conflicts among actions. For example, consider an instantiation of the build operator that involves changing one source module. The new source module must be preserved from the time it is created until it is checked-in. An action such as editing (that would contribute to building the next system version) cannot be allowed to interfere with accomplishing the check-in. Strategies for salvaging the plan include preventing editing until check-in occurs or inserting an action to make another copy of the module.

Conflicts can occur within a single plan as well as between two plans being carried out concurrently. The simple action of deleting something has the potential of interfering with almost any active plan. Conflicts can also occur between what the programmer is doing and what the intelligent assistant is doing. If the intelligent assistant is to achieve a

particular goal, it must do so without interfering with any in-progress activities of the programmer. (Obviously, conflicts cannot be anticipated and prevented if the intelligent assistant is not tracking what the programmer is doing; this confirms the importance of providing plan recognition in addition to planning in the intelligent assistant architecture.)

It is important to note that conflicts cannot be identified simply by querying the state of the world—they can only be identified through knowledge of the plans that are currently in progress. Plans show *why* certain states of the world hold, and thus *how long* they must be held. For example, once a subgoal is satisfied, it should remain satisfied at least until all its sibling subgoals are also true (at which time the operator completes). If some action disturbs this subgoal, then the operator cannot complete until the subgoal is re-achieved. This means that plans describe not only the significance of actions, but also the significance of various aspects of the state of the world.

Planning algorithms handle conflicts via domain-independent methods. Associated with each precondition or subgoal is a *protection interval* that begins when the precondition or subgoal is achieved and ends when the operator begins (for preconditions) or ends (for subgoals). A conflict occurs when a precondition or subgoal is destroyed during its protection interval. Domain-independent methods for resolving conflicts, described in [Sacerdoti, 1977; Tate, 1976; Wilkins, 1984; Chapman, 1987], include imposing additional temporal constraints and selecting alternative operators.

This ability to anticipate and prevent conflicts (among operators that are fundamentally *if-then* rules) distinguishes planning from other rule-based systems. Marvel [Kaiser & Feiler, 1987] uses forward and backward chaining to automate chores that are prerequisites to or consequences of programmer actions. Glitter [Fickas, 1985] cooperatively automates goals in the transformational development process. These systems do not prevent conflicts among rules. Systems based on <*condition,action*> rules, such as CLF [Balzer, 1987] and

Genesis [Ramamoorthy et al., 1985], lack descriptions of effects; they cannot reason about the consequences of actions, and therefore cannot prevent conflicts. Planning techniques are used in two existing systems. Agora [Bisiani et al., 1988] provides a domain-specific planner for tasks relating to heterogeneous, parallel systems. Although help systems usually provide advice that is too local to qualify as process support, UC [Wilensky et al., 1984] now uses planning to gain a more global perspective [Luria, 1987].

1.4 A Research Issue in Plan-based Intelligent Assistance

Any architecture for intelligent assistance, whether plan-based or not, must be assessed on its overall usefulness: is the assistance provided of a substantive nature? Error detection and correction, cooperative automation, and maintenance of agendas/summaries are all useful *forms* of support (and one advantage of a plan-based approach is that it provides all these forms within a single paradigm). But if automation can only be achieved after extensive user input, if only a small percentage of potential errors can actually be detected/corrected, if some normally available options are precluded, if numerous exceptional situations simply cannot be handled, then there is only an illusion of meaningful support: the *content* of the support is not substantive. In the worst case, the intelligent assistant may even be counter-productive.

1.4.1 Knowledge Representation for Substantive Support

Substantive support requires that significant amounts of domain knowledge reside with the intelligent assistant, not just with the user. While we accept, from the outset, that the level of knowledge in the intelligent assistant will not equal that of the user, more knowledge gives the intelligent assistant more independence and autonomy: the ability to plan automatically with less guidance from the user, and the ability to critique, rather than merely accept, the plans of the user. If too little knowledge is invested in the intelligent assistant, then the full potential of a plan-based approach is not attained. In that case, a

similar level of support can undoubtedly be achieved more economically by approaches other than planning.

In a plan-based intelligent assistant, domain knowledge resides in the operator definitions. The depth of knowledge that can be captured is a function of the expressive power of these definitions. The level of knowledge diminishes, and the goal of substantive support recedes, with each parameter that is insufficiently constrained, each restriction on the applicability of an operator that cannot be expressed, each activity that must be treated as "optional", each exceptional situation that cannot be handled, and each type of failure for which there is no recovery strategy.

The research issue addressed in the remainder of this dissertation concerns the expressive power of operator definitions in a planning formalism: what kinds of domain knowledge are needed, what representational limits are reached, and how can the limits be overcome? While this issue is central to achieving the kind of intelligent assistance we have described, it is not unique to this application; rather, it is relevant to all applications of planning and plan recognition technology.

1.4.2 Scope of Research

In this dissertation we identify two areas where the expressive power of operators is limited, and we develop extensions to address both limitations; one extension involves nonmonotonic reasoning and the other involves metalevel knowledge. (In order to show that these extensions respond to real limitations in existing mechanisms, we describe in some detail the types of knowledge that can be expressed in standard operator definitions.)

The first extension presented in this dissertation establishes a critical role for a special type of domain knowledge: knowledge about what is typical or expected given a certain context, as opposed to knowledge about what is certain in that context. While definitely of

secondary importance under normal conditions, this empirical knowledge can play a key role in applications, like intelligent assistance, that involve *hidden state* (i.e., where information about the state of the world is incomplete). When state information is missing, it is impossible to exploit domain knowledge needed to evaluate some preconditions, subgoals or constraints in operator definitions. This means that the autonomy of the intelligent assistant, indeed the very ability to represent the domain in a nontrivial way, is at risk. The solution to acquiring additional state information is to enable plausible assumptions about the missing values to be drawn from the observed state. The process of making these assumptions can be formalized through nonmonotonic reasoning, which is then integrated into the planning system.

The second extension presented in this dissertation involves domain knowledge that is directly concerned with plans themselves. Planning algorithms already incorporate general knowledge about how to *construct* plans, given knowledge about actions (from the operator definitions). But some domain knowledge is not directly concerned with actions; it has more to do with plans—when/how to *modify* them. For example, if a subplan for a specific goal fails to achieve that goal, continuing with the overall plan may involve replacing the original goal by another (usually simpler) goal. Or, under selected circumstances, an additional subgoal may be needed or a subgoal may be dropped or modified. We show how this kind of knowledge, which concerns transformations on plans, can be formalized as metaoperators—actions that operate on plans.

These two extensions build upon the two methods of knowledge representation already implicit in a planning system, where domain-independent knowledge about constructing plans to achieve goals (as described in section 1.3.3 above) is already built into the planning/plan recognition algorithms and basic knowledge about a specific domain is captured in the operator definitions. This results in four separate mechanisms, each with distinctive representational capabilities, for capturing process knowledge.

The emphasis of this dissertation is on attaining the appropriate level of representational power in a planning system in order to reason about user actions. Providing appropriate representations for domain knowledge does not solve the problem of actually acquiring that knowledge. One method of dynamic incorporation of new domain knowledge is described in [Broverman & Croft, 1987]. There is an opportunity to apply the techniques of *knowledge acquisition* to improve the process of identifying and formalizing domain knowledge; the work of [Lefkowitz, 1987] addresses this problem. Also, reasoning about actions (so as to have something interesting to communicate to the user) does not directly solve the problem of deciding when or what to communicate and actually doing the communication. There is an additional need to design a suitable interface between user and intelligent assistant that will facilitate communication; one approach is described in [Mahling & Croft, 1988], another in [Wilensky et al., 1984]. Perspectives on this issue are provided by work in human factors, intelligent interfaces, natural language interfaces, and man-machine interaction. The roles of knowledge acquisition and user-interfaces in intelligent assistance are separate research topics unto themselves, and are not addressed here.

The (intelligent assistant) domain from which examples of complex knowledge are drawn is the software development process. Examples are chosen to elicit representation issues in planning formalisms; they are not intended to cover software process issues in a complete or uniform way. Naturally, the examples do shed light on the organization and content of software processes—another research issue in its own right—but this is not their primary purpose.

Although many interesting issues in intelligent assistance involve project-level coordination, cooperation, and negotiation among multiple project members, we restrict our attention to supporting an individual programmer from a local perspective. Although the local perspective on activities will necessarily capture certain global considerations—such

as project policy and use of tools like RCS that assist with coordination—there are limits on what can be covered from the local perspective alone. Both the local and global perspectives need to be addressed, but each one is sufficiently difficult to make a separation advisable. Note that an advantage of the plan-based approach is that there is already a body of results on planning by and for multiple agents.

While the intelligent assistant architecture is based on an integration of planning and plan recognition, this dissertation emphasizes plan recognition. (Planning is covered only to the extent that it serves a plan recognition function, as discussed in sections 2.2.2.5 and 4.1.2.2.2; however, this use is sufficient to show that a planner can in fact operate in the context of partial plans constructed by a plan recognizer.) To cover both planning and plan recognition algorithms in equal detail is beyond the scope of a single dissertation.

1.5 Contributions

In this section we present the six contributions of this dissertation research.

The first contribution is the conceptual architecture for intelligent assistance, already presented in section 1.2, that is based on the integration of planning and plan recognition. This architecture is a successor to an earlier effort aimed at developing intelligent assistance through an event-based description of user activities [Huff & Lesser, 1982; Croft et al., 1983; Croft & Lefkowitz, 1984]. The primary objective of this new architecture is to support reasoning about actions from a process perspective; this reasoning is necessary to achieving assistance that includes automation, but allows for other types of support when total automation is impossible. The key aspects of this architecture, as described in section 1.3, include domain-independence, mixed initiative interaction at multiple levels of abstraction, and the explicit representation of goals that allows plans to be tailored to current context.

The second contribution arises from the observation that the intelligent assistant application involves hidden state, where the missing state information is required in order to be able to represent the domain in a nontrivial way. The key insights leading to a solution are that it is possible to compensate for missing state information by using additional domain knowledge to make plausible assumptions and that this is an application of nonmonotonic reasoning. We present the conceptual approach as well as the technical details for accomplishing this in plan recognition. There are two parts to the approach: implementing the nonmonotonic reasoning and integrating it with the plan recognizer. Implementing the nonmonotonic reasoning involves selecting an appropriate nonmonotonic system as well as showing how it is used to capture two types of domain knowledge: empirical knowledge and fundamental domain principles that cannot be exploited without the empirical knowledge. The introduction of assumptions, which may turn out to be incorrect, affects how the plan recognizer generates and compares alternative interpretations. In addition, the plan recognizer must identify situations in which the assumptions are incorrect (or probably incorrect), and be able to revise the assumptions accordingly in order to proceed with the recognition task.

The third contribution is a single conceptual framework for capturing additional domain knowledge that is directly concerned with plans. This contribution is based on the observation that some domain-specific knowledge is most naturally expressed as transformations on plans: when to substitute one goal for another, when to add an additional subgoal, how to repair a plan when an operator fails, or when to instantiate a new top level goal. A solution is based on the insight that since these transformations act upon plans as the state of the world, they can be formalized as metaoperators/metaplans. This extends the role of metaplanning, which has previously been used to capture domain-independent knowledge about the construction of plans. A single formalism, metaoperators, is used in lieu of adding many special-purpose constructs to the operator

definition language. This contribution is conceptual in nature, as an implementation of metaplanning is not included in the GRAPPLE system that serves as the proof of concept for the other contributions.

The next three contributions are secondary contributions, arising out of the context of the first three (primary) contributions.

The fourth contribution arises out of the use of nonmonotonic reasoning, as already mentioned, and the concomitant need to revise assumptions when they are found to be incorrect. We present a new method for revising assumptions, called *reconciliation*, that achieves a higher degree of integration of new information into the existing knowledge, and we give an algorithm for it. Reconciliation attempts to explain why assumptions were incorrect, in addition to working out the consequences of making a correction. The identification of potential explanations is based on an additional application of nonmonotonic reasoning.

The fifth contribution involves an extension of the concept of integrating planning and plan recognition. We show how planning, in the standard sense of constructing a plan to achieve a goal, serves an important function in accomplishing plan recognition, in the standard sense of identifying the goal (and plan) that a group of actions is intended to achieve. This use of planning as an integral part of a plan recognition algorithm solves a collection of technical obstacles inherent to the incremental recognition of hierarchical plans.

The sixth, and final, contribution is a body of specific examples of software processes. While it is generally recognized that there are software processes, there is currently not a great deal of understanding as to what those processes actually are. Thus, the examples describe, and give names to, processes—archiving, releasing, promoting, building, testing, checking-in, etc. In addition, the examples are interesting for the specific

way that these processes are defined. The examples show how processes can be organized into hierarchical levels and modularized (into separate subgoals) at a given level. They also show special techniques that may be needed when defining processes formally (for example, a distinction is made between a file, which is simply a place to store things, and contents, which are the things such as source text that can be stored in files; this distinction is needed in file-based environments like UNIX to define certain actions such as those that replace the current contents of a file with some other contents.)

1.6 Proof of Concept

Intelligent assistance based on a global perspective towards user activities is predicated upon the existence of a definable structure to those activities. The claims that this structure exists and that it is hierarchical in nature are borne out by the examples already given in this chapter, and will be further supported by additional examples in later chapters. A plan-based approach to intelligent assistance is further predicated upon the notion of *goals* as a central concept in representing these structures. Again, the claim that activities are directed at achieving definable goals is proven through specific examples of goals, both in this chapter and throughout the remainder of the dissertation. Since all the examples are taken from the software development process application, these claims are obviously limited to that domain; however, the office automation domain is addressed in [Croft & Lefkowitz, 1988], which presents examples of hierarchical office procedures and goals.

The GRAPPLE system implemented in Prolog and running on a Macintosh II™ was developed specifically as the principal proof of concept for this dissertation research. GRAPPLE is a prototype for a plan-based intelligent assistant, emphasizing support through plan recognition (it also contains a planner as described in sections 2.2.2.5 and 4.1.2.2.2). It is domain-independent, tied neither to the software development application

---

Macintosh is a trademark of Apple Computer Corporation.

nor to any specific software environment, toolset, or process. The GRAPPLE system consists of approximately 7,000 lines of Prolog.

GRAPPLE accepts an arbitrary operator library (written in the operator definition language described in section 2.1); these operator definitions are analyzed, checked for errors, and stored in an internal form. Then, given an initial state and a stream of actions (representing the primitive operators in the current library), GRAPPLE performs plan recognition. It does the necessary simulation to model the successive states of the domain world using the predicates defined in the operator library. The results of plan recognition are then used to provide intelligent assistance. The user is notified when GRAPPLE cannot recognize an action or when a protection violation is detected. The user can also ask for current status, which results in a display of the current plan(s) in progress, showing what has been accomplished and what remains to be done at the different levels of abstraction.

Through a color graphics interface, the operator library, the state of the plan, the state of the world, and the state of plan recognition are displayed as trees in scrollable windows, and can all be examined. This interface is designed to show the internal workings of the plan recognition process and emphasizes how reasoning about actions is accomplished; considered as a prototype user-interface for an intelligent assistant, it is extremely rudimentary (that is, it is not the product of a serious endeavor to address how the results of reasoning about actions are best communicated to the user).

The GRAPPLE implementation serves two purposes. First, it shows that the approach to plan recognition described in section 2.2 and that the extension to plan recognition in the presence of hidden state described in chapters 3-4 are more than descriptions on paper—they have been developed in sufficient detail to have been implemented. Second, it shows that the examples used in this dissertation, which consist of sets of operator definitions for various aspects of software activities, are more than

sketches—they have sufficient completeness, consistency and precision to have been run on the GRAPPLE system.

Because GRAPPLE simulates the state of the world rather than being integrated into an actual existing environment, it does not address the practical problems this integration entails, nor does it show that these problems can be overcome. However, the Marvel system [Kaiser & Feiler, 1987] has solved these problems for the generic UNIX environment; since UNIX contains no explicit environment database in the sense of Figure 1.6, Marvel has specifically addressed the issue of maintaining such a database as actions are executed. While there are significant differences between Marvel and GRAPPLE in how rules/operators are used to model software processes and in the nature of the assistance provided, the two systems are sufficiently similar at the environment interface that the Marvel interface approach should transfer to GRAPPLE.

It should be re-emphasized that the thrust of this dissertation concerns the development of appropriate mechanisms for representing complex domain knowledge in planning formalisms. Therefore the examples of software processes are chosen to highlight representation issues. Given this thrust, no explicit proof is offered that the software process examples given correspond to (parts of) actual software processes that might be carried out on an actual project. However, the examples were drawn from observations of the processes in use on actual projects and some informal study of transcripts of terminal sessions. Therefore, we do believe that the examples are reasonably well-grounded in reality, and that this will be apparent to those who are familiar with the software development domain.

1.7 Guide to Dissertation

The remainder of this dissertation consists of five chapters and six appendices, organized as described below.

Chapter 2 describes a baseline operator definition language and the plan recognition algorithm that implements it. The purpose is to show the kinds of domain knowledge that can be accommodated in traditional operator definition format without raising new research issues. The operator definition language includes some features not previously supported in planning systems, but these features represent engineering extensions. They reflect certain domain characteristics, common to the application domains of intelligent assistance, that are not encountered in more traditional applications of planning (such as the blocks world, equipment repair, and robot navigation). This chapter also includes a description of the process by which domain knowledge is translated into operator definitions.

Appendices A-C support the material in Chapter 2. Appendix A consists of an example operator library consisting of 15 operators for the "setup-environment" task; this task is used throughout Chapter 2 both to illustrate operator definition language features and to illustrate plan recognition issues. Appendix B summarizes the plan recognition algorithm described in section 2.2. Appendix C consists of examples of plan recognition in action; three scenarios, each consisting of a sequence of actions for setting up an environment, are traced step-by-step through the plan recognition algorithm.

Chapter 3 addresses the problem of achieving a nontrivial representation of a domain that involves hidden state; in the absence of complete state information, an intelligent assistant has no basis for independently critiquing many actions of the user, and thus its autonomy is at risk. The solution to acquiring additional state information is to use domain knowledge to make plausible assumptions about the missing values using the *observed* state, a process which is formalized as nonmonotonic reasoning. Using these plausible assumptions, the *credibility* of competing alternatives can be compared independently of the user; actions that are consistent with current assumptions will have the highest credibility. If it becomes necessary, actions that have low credibility can still be pursued after *reconciling* the assumptions with the requirements of the actions. This chapter contains a

complete description of this approach to deeper domain modeling, additional examples for the software process application, and a detailed discussion of translating domain knowledge into nonmonotonic/monotonic form.

Chapter 4 describes the implementation details of the approach to deeper domain modeling presented in Chapter 3. There are two major topics. The first concerns the technical and algorithmic details of marrying nonmonotonic reasoning (about the state of the world) with a plan recognizer. This includes the truth-maintenance system (TMS) used to implement nonmonotonic reasoning, changes and additions to the basic plan recognition algorithm, the calculation and use of credibility, and the interface to reconciliation. The second topic is reconciliation, a new approach to revising assumptions in a TMS. The motivation for a new approach is given, the approach is described, and then an algorithm for reconciliation is given.

Appendices D-F support the material in Chapter 4. Appendix D describes the TMS implementation used to support nonmonotonic reasoning. Appendix E gives a revised algorithm for plan recognition that includes the modifications needed to accommodate the deeper domain modeling. Appendix F describes the algorithm for reconciliation that is used to revise assumptions about the state of the world.

Chapter 5 presents the use of metalevel constructs to capture domain-specific knowledge. Examples are used to motivate the need for a metalevel representation. These include knowledge about recovery strategies when operators fail, domain-dependent strategies for preventing conflicts from destroying plans, shortcuts that are not always safe, and other types of context-dependent variations on operators. We show how the knowledge in these cases takes the form of transformations acting on plans and the operator instances in the plans. Then we show how these transformations can be formalized in

metaoperators, using the operator definition language of Chapter 2 to define actions on the plans. Finally, we discuss some of the implementation issues in a metaplan approach.

Chapter 6 summarizes the results of the preceding chapters, presents overall conclusions, and discusses open issues suitable for further investigation.

CHAPTER 2
FOUNDATIONS:
OPERATOR DEFINITIONS AND PLAN RECOGNITION

In this chapter we present a basic language for operator definitions and a basic algorithm for plan recognition, in order to establish a clearly defined foundation on which the research described in the following three chapters is built. In the first section of this chapter, a software development task is used to highlight specific language issues that arise in operator definitions for a typical set of software activities and to showcase one set of language features designed to handle these problems. In the second section, a plan recognition algorithm is presented that is appropriate for the intelligent assistance applications and that supports these language features. A novel feature of this algorithm is the use of a planner in support of plan recognition functions.

2.1 Operator Definition Language

The applications for intelligent assistance are all complex domains—at least as complex as any in which planning has been applied. Early planning applications were necessarily simple, and the operator definitions made appropriate simplifying assumptions to concentrate on the initial theoretical challenges. For example, NONLIN [Tate, 1976] did not allow variables in operators; STRIPS [Fikes & Nilsson, 1971] assumed that all

variables appeared in, and therefore were bound through, the goal statement. Such assumptions were later relaxed in systems like MOLGEN [Stefik, 1981a,b] (with sophisticated constraints on variables) and especially SIPE [Wilkins, 1984], which took seriously the issue of engineering a realistically useful operator language.

In addition to simplifying assumptions made to focus on theoretical issues, the operator definition languages documented in the literature were often designed around examples from a single domain (SIPE is an exception). This led to selective development of language features, since different domains raise different representation issues; some domain characteristics have simply not been explored. For example, when the domain does not involve dynamic creation and destruction of domain objects, then there is no need for language features that support this function.

Certain kinds of domain complexity can be handled in operator definitions by designing language features that extend the basic framework of state-based operator definitions without raising new research issues. In this section, we explore, by way of an example, a set of baseline language features required in intelligent assistance for software development. (The need for these features is not confined to the software domain, but extends to other intelligent assistant applications.) First, we present an example of an operator library, and then we discuss some of the special language features used in the example.

2.1.1 An Example Operator Library

In this section we give an example of a library of operator definitions for a task from the software process. Although this task is only a very small part of the software process, it is representative of the complexity in this domain. First, the task is described briefly; then the creation of operator definitions for the task is discussed.

2.1.1.1 Informal Description of Setup-Environment

Consider the activities preliminary to compiling/linking and testing a new version of a software system. The new system is first defined, described, or specified in some fashion. Then the programmer sets up an appropriate environment in which to apply the building and testing tools; this is the set of activities to be described in detail. After the environment is setup, the building and testing can take place.

Setting up an appropriate environment can be done in several different ways. We describe one approach, commonly used in directory-structured operating systems like UNIX. All the compilation units representing the baseline from which the new system version is to be developed are gathered into a "reference directory". A "working directory" is designated to hold all the new compilation units unique to the new system version. This arrangement of files is made on the assumption that the tools (e.g., compiler, linker) first search in the working directory, and then fill out their remaining needs for compilation units from the reference directory. When two (or more) new system versions are being developed from the same baseline, they can share one reference directory, but each system version always needs its own working directory. Obviously, one directory cannot serve as both a reference and a working directory.

In this approach, setting up an environment consists of three main tasks, which can proceed concurrently. They are shown informally in Figure 2.1, as setting up the reference directory, setting up the working directory, and making the components unique to the new system. Setting up the reference and working directories break down into further subtasks: the programmer has to commit an appropriate directory, empty it of extraneous files, and fill it with the relevant files (e.g., files containing baseline system components for the reference directory, and newly edited components for the working directory). These tasks are accomplished with operating system commands to make directories (*mkdir*), copy files

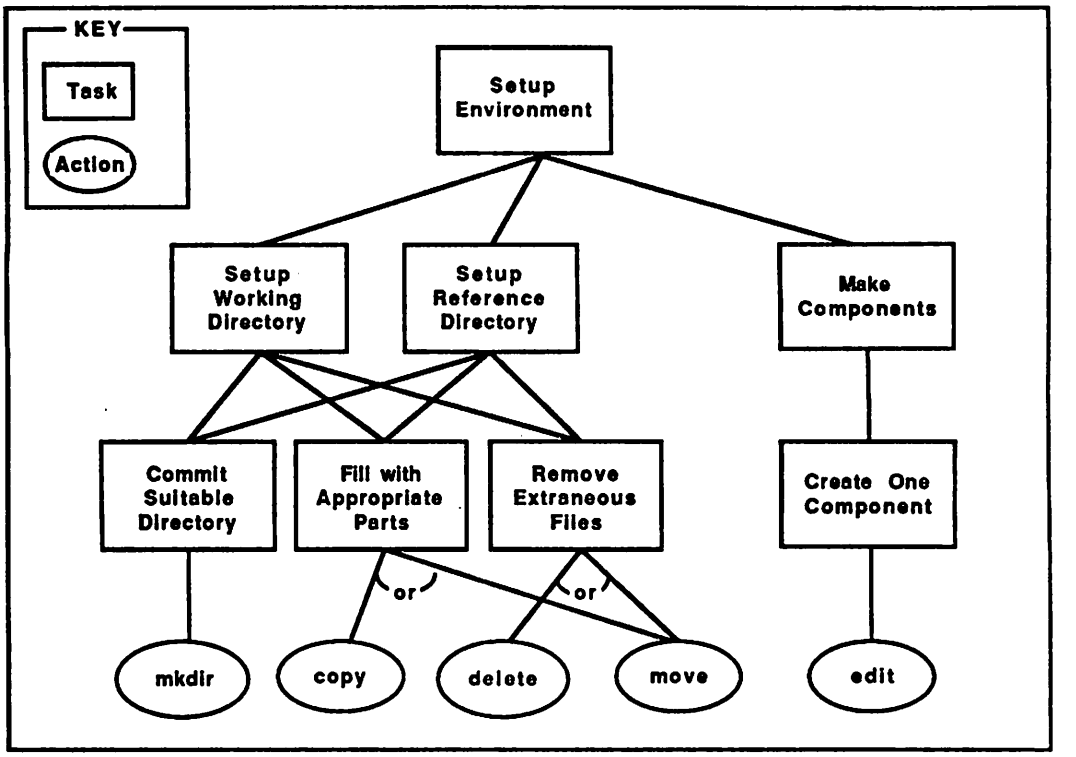

Figure 2.1 Task Breakdown for Setup-Environment

(*copy*), move files (*move*), and delete files (*delete*). Making the system components breaks down into a subtask to create a single component; this subtask may be repeated if more than one new component is needed. It is accomplished by (possibly repeated invocations of) the command to edit existing or new files (*edit*). (In this view of the tasks, there is a kind of "producer-consumer" relationship between making the system components and placing them in the working directory.)

The state schema that is relevant to this example includes four types of entities: directories, files, contents of files, and systems. An initial sketch of this state schema is shown in Figure 2.2. Files are in directories, contents are stored in files, contents are of different kinds (such as source, include, executable), contents can be part of systems, (other) contents represent the executable forms of systems, and systems can be baselines to

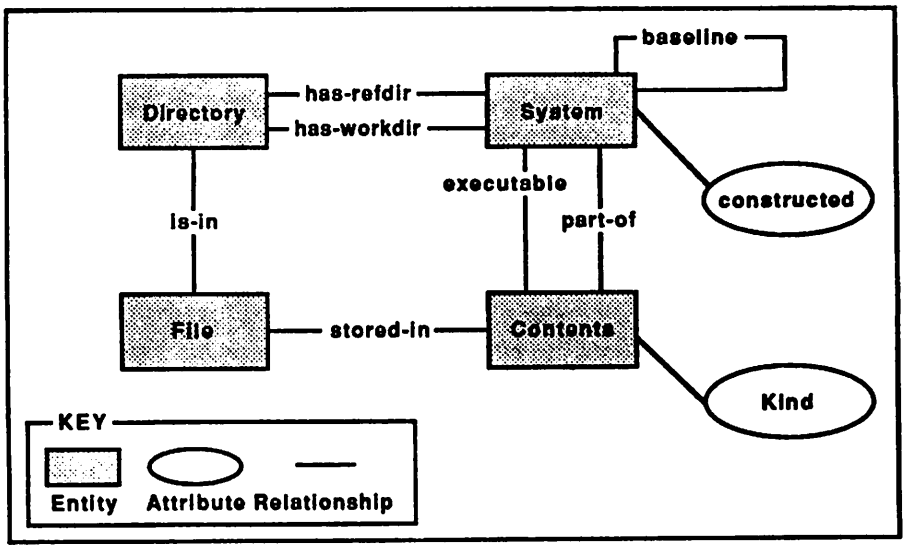

Figure 2.2 State Schema for Setup-Environment

one another. Directories can be used by systems as reference or working directories. Systems probably have a variety of attributes, including whether or not they have been constructed (here taken to mean built and unit-tested). Note that directories, files and their contents have concrete counterparts in the software environment, whereas systems are conceptual entities that are part of a programmer's view or interpretation of that environment.

In order to complete the example (and to support later discussion of plan recognition), these setup activities need to be placed in context. But to keep the examples to manageable size, it is necessary to abstract the surrounding activities, rather than represent them in full detail. Whatever the process of defining/specifying a new system version to be developed, it always results in the creation of a new system, which has not yet been "constructed". So we assume that *define-system* is a primitive operation that accomplishes just that. Whatever details are involved in compiling, linking and testing, the net results (assuming success of course) are that there is a new executable that represents the system, that the

reference and working directories are "released" from use by this system, and that the system is considered "constructed". Again, we assume a primitive operation *construct* that represents all this. And, as a final simplifying assumption, we hide all the detail of individual invocations of the editor to create new system components in a single primitive operation called *create* that makes a component*.

2.1.1.2 Characteristics of the Example

This example was chosen for four reasons. First, it is a real example; we have observed programmers using this type of directory structure in their work. Second, it uses all the interesting operator definition language features that we are about to discuss. Third, it is an ideal example to show the strengths of a planning approach in the area of tailoring plans to context. Fourth, it is a fairly compact example (the formal definition involves 15 operators.) The example was not chosen because it is an important part of software process, or because it universal to processes. It was chosen to showcase the types of issues that arise in formally defining software processes.

The example shows four characteristics that we have claimed are typical for processes. First, there is an obvious hierarchical structure, as suggested in Figure 2.1. Second, there are many different action sequences that can be used to achieve a given goal. In some cases, there are several operators to choose from. For example, reading off Figure 2.1, *fill* can be achieved by the use of *copy* or *move,* or any combinations thereof. In other cases, the actions chosen depend on the context—whether there are existing, uncommitted directories, or the current placement of files in those directories, for example. Third, there is a high degree of interleaving of (sub)tasks. For example, the programmer can intermix

---

* Extending the example is principally a matter of supplying subgoals in the operator definitions for *define_system*, *create*, and *construct* and supplying definitions for additional operators that will achieve these subgoals.

actions to setup the reference directory with actions to setup the working directory, or can intermix actions from two separate setup-environment plans. Fourth, there are opportunities for conflicts to arise that will destroy the effectiveness or efficiency of the plans. If the reference directory is already setup, and the programmer is now setting up the working directory, then there are better ways of emptying the working directory of extraneous files than moving them into the reference directory; this will destroy the "readiness" of the reference directory, meaning that more work will have to be done to reestablish "readiness". Such a conflict leads, at best, to an inefficient plan; if not detected, it leads to an improper plan (if the file is never subsequently moved out of the reference directory).

2.1.1.3 Formalizing the Definition of Setup-Environment

The complete operator and state schema definitions for the setup-environment example are given in Appendix A. In this section, we describe the process by which these definitions were created. There are two basic forms of operators: primitive and complex. We start with the primitive operators, and then discuss the complex operators.

2.1.1.3.1 Writing Primitive Operators

For each primitive operator, we write an operator definition, enumerating the goal, preconditions, constraints, and effects of the operator. There are seven primitive operations to be defined: *mkdir, copy, move, delete, create, define-system,* and *construct.* These represent the actions that are available to be executed. *Create, define-system,* and *construct* are, as mentioned previously, actually abstractions covering what would otherwise be multiple operators. In fact, the precondition to *construct* that requires that the environment be setup is where all the interest is in this example.

Consider writing the *delete* operator (replicated from Appendix A in Figure 2.3). Start with the effects, where the changes that the *delete* operator brings about are defined. These changes, so "obvious" that they are taken for granted, are that the file will no longer be in the directory it was in, that the contents of the file will no longer be stored in the file, and that the file will no longer be in existence (this is a predefined predicate, described further in section 2.1.2.1). When *delete* is actually executed, only the file name is given as a parameter. Since we have to know the contents of the file and the directory containing the file to accomplish these effects, we add constraints to "bind" these additional parameters. Only one precondition appears to be relevant: that the file exist. (This precondition is static; discussion of static versus dynamic preconditions appears in section 2.1.2.3).

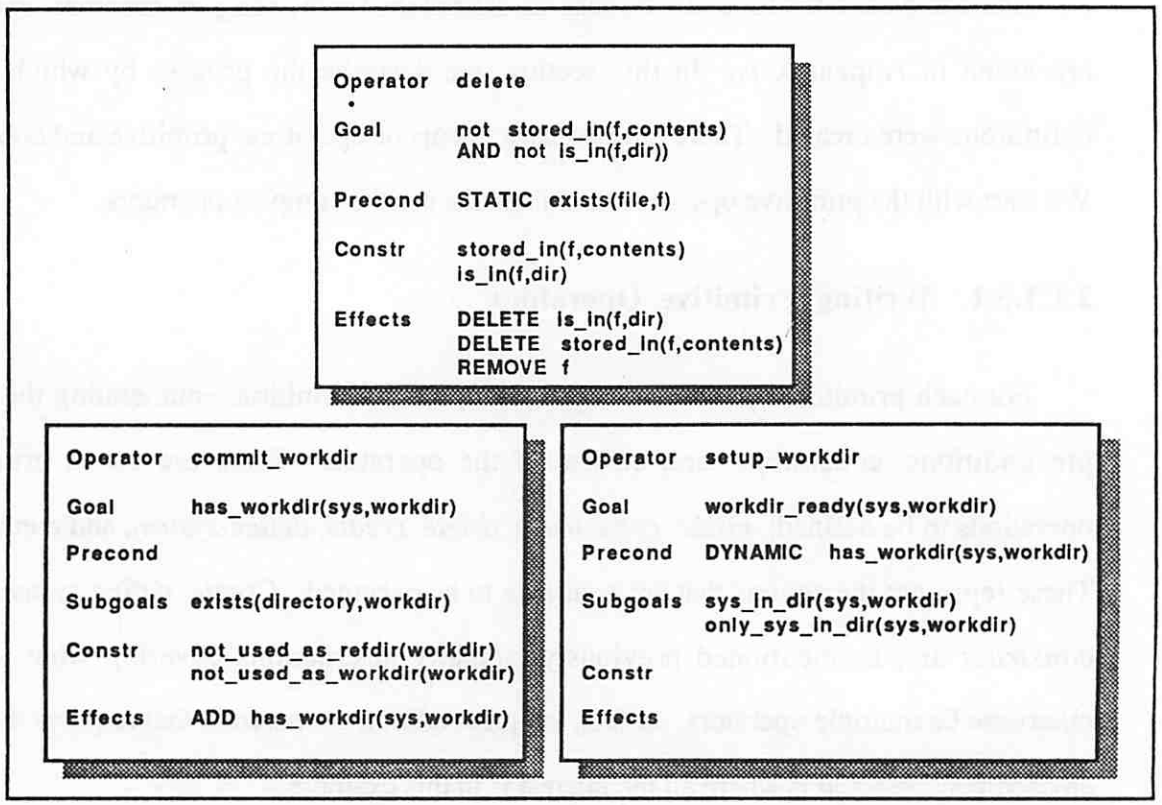

Figure 2.3 Example Operator Definitions for Setup-Environment

Finally, the goal is written by considering which of the effects are "main" effects of interest to other operators that will be in the operator library. Given our intentions about this operator library, deleting files is going to occur because we want the file out of some directory in order to insure that the contents are not represented in that directory. Therefore, we omit *not exists(file,f)* from the goal, although there would be no harm in including it.

2.1.1.3.2 Writing Complex Operators

The task breakdown in Figure 2.1 suggests the candidate complex operators: *setup-env, setup-refdir, setup-workdir*, and *make-components*. Furthermore, the three activities of acquiring a directory, emptying a directory of extraneous files and filling it with relevant parts (all needed both by *setup-refdir* and *setup-workdir*) give three more complex operators: *commit-directory, place-parts, remove-extra-files*. The latter two are complex operators because they could involve moving multiple files around; therefore, they are tasks that cannot necessarily be accomplished directly by the execution of a single primitive operator. Actually, the task of committing a directory must be split into two distinct tasks: *commit-workdir* and *commit-refdir*; this is because different usage restrictions apply—a reference directory can already be in use as a reference directory (provided the baseline is the same), while a working directory cannot. These must be complex operators because, although accomplished by a single execution of *mkdir*, they serve to give additional meaning to the *mkdir* operation as described below.

Consider writing the operator definition for *setup-workdir* (replicated from Appendix A in Figure 2.3). The goal of *setup-workdir* is that a working directory is ready for development of some system. We don't have a predicate for this, so we define one called *workdir-ready* with two arguments, a system and a directory. Reading off the task breakdown in Figure 2.1, we see three subsidiary activities: committing the directory,

placing the new parts, and removing extra files. Committing a working directory for use by this system should be done first—it becomes the precondition; the two remaining activities become the subgoals. If the precondition is satisfied, then a working directory is committed to this system; we have already named this predicate *has-workdir(system,dir)* (see Figure 2.2). Additional predicates are needed to express the subgoals. So, we define *sys-in-dir(system,dir)* and *only-sys-in-dir(system,dir)* for the two subgoals. (More discussion on designing predicates appears in section 2.1.2.2.) We can leave all constraints on appropriate directories to the operator that achieves the precondition. As to the effects, none are needed. The goal, *workdir-ready*, is going to have to be a computed predicate (see section 2.1.2.2), and computed predicates are never added or deleted explicitly; their truth value is "computed" when it is needed.

Commit-workdir (replicated from Appendix A in Figure 2.3) represents another type of complex operator. The goal of *commit-workdir* must be the predicate *has-workdir(system,dir)*. As confirmation, we see that it matches the precondition of *setup-workdir*, just as we intend. The task breakdown of Figure 2.1 shows one subsidiary activity: creating a directory. This becomes the only subgoal of *commit-workdir*; the expression for this subgoal is the predicate *exists(directory,dir)*; this predicate is predefined (see section 2.1.2.1 for further information). There are no preconditions. The constraints cover the restrictions that apply: the directory cannot be in use either as a reference or working directory; these are again new predicates. Unlike *setup-workdir*, *commit-workdir* has effects; generally, a complex operator should have effects if the goal of the operator is not logically implied by the conjunction of the subgoals, (except when the goal involves a computed predicate, as described in section 2.1.2.2 below). The existence of a directory does not logically imply that is being used as a working directory; it could be a reference directory, or it could have some other use if our example were all-encompassing.

Therefore, adding the *has-workdir* relationship must be an explicit effect of *commit-workdir*.

2.1.1.3.3 Task Structure and Domain Predicates, Revisited

During the operator definition process we have introduced new predicates, and changed the task breakdown slightly. All the predicates are given in the complete world state schema in Figure 2.4. The revised task breakdown structure is given in Figure 2.5. The changes include adding (the abstractions) *define-system* and *construct*, and simplifying the structure beneath *make-components* using *create*; also, the original task of committing a directory has been split into two separate tasks as described earlier. Figure 2.5 shows all the operators and their gross interrelationships. Figure 2.6 gives additional detail; it

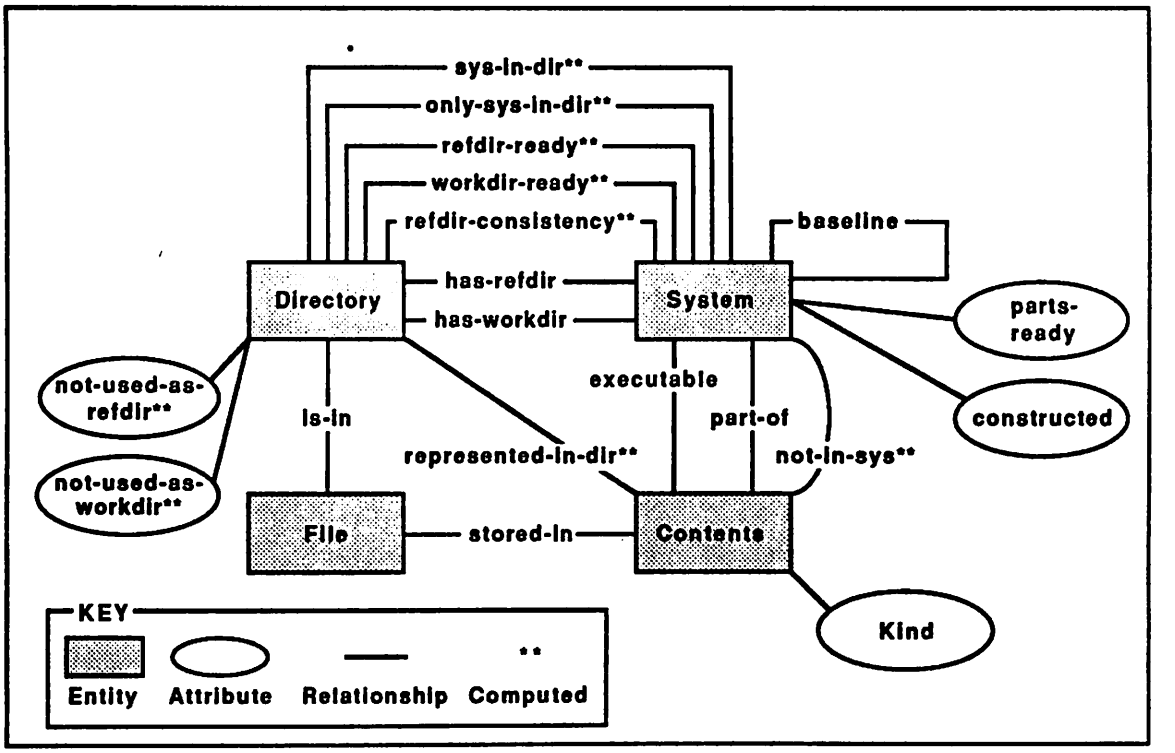

Figure 2.4 Expanded State Schema

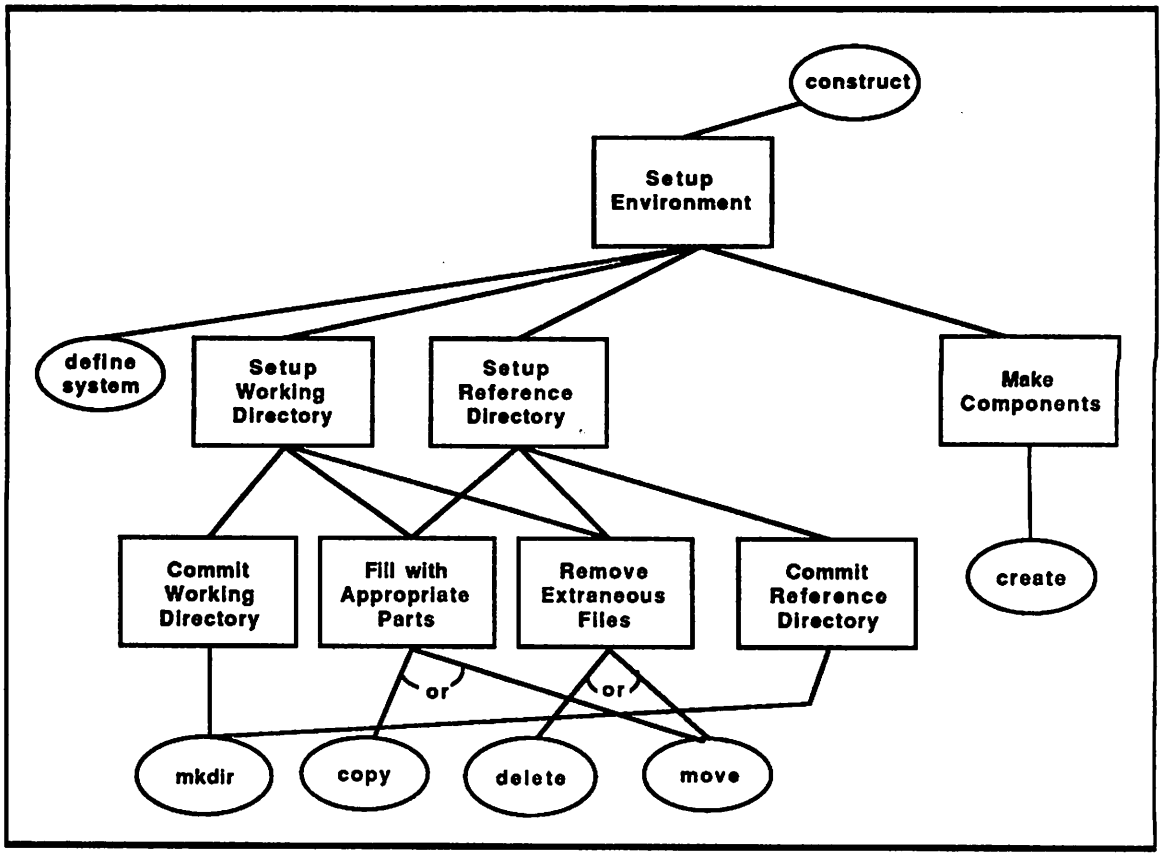

Figure 2.5 Expanded Task Breakdown

includes the internal structure (preconditions and subgoals) of each operator. In this figure, there are no direct relationships between operators: only between a lower operator and a precondition or subgoal of a higher operator that the lower operator could be used to achieve. (This is confirmation of the emphasis of planning on goals to achieve, not actions to perform.) In fact, these "achiever" relationships are not given explicitly in the operator definitions; they are computed as described in section 2.2.2.1.

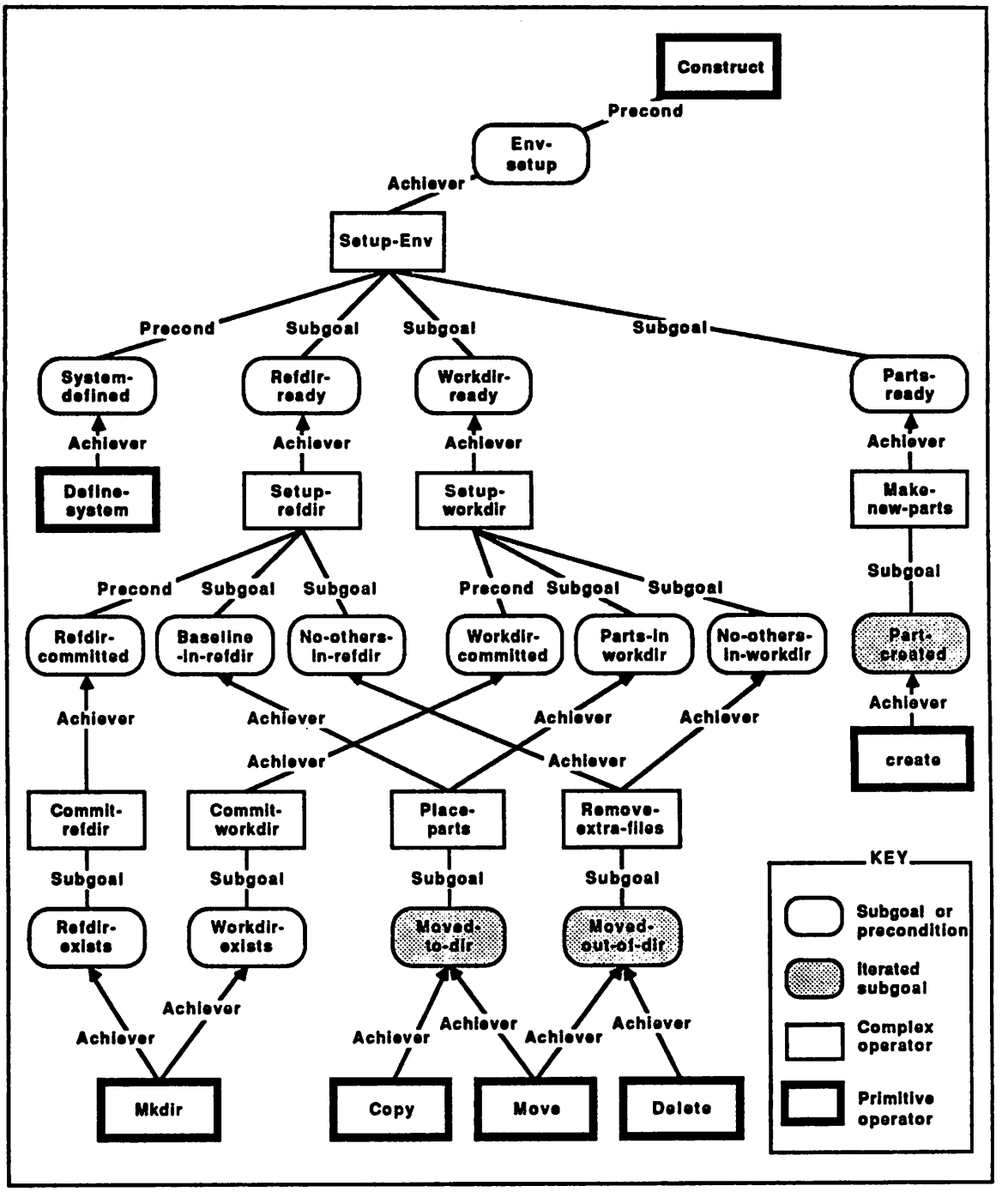

Figure 2.6 Operator Library Overview

2.1.2 Operator Definition Language Features

In section 2.1.1, we concentrated on the process of translating software development knowledge into operator definitions, and omitted discussion of some of the special features of the operator definition language. These details are covered in this section. In some cases, similar features have appeared in other planning systems, while other features have been recognized as useful but not previously designed or implemented. Engineering an operator language is not a matter of plucking the best features from previous systems and then adding a few new ones. Decisions must be coordinated because the features interact with one another. One such coordinated set of decisions is presented below; it represents only one of many possible solutions to language design.

2.1.2.1 Dynamic Domain Entities

The software development process, like the other applications of intelligent assistance, is basically a constructive.process: domain entities (such as directories, files and their contents, and systems) are created dynamically, and some eventually get deleted. Most traditional applications of planning have been in static domains, where actions do not create new entities. Creation and deletion are handled by supporting two new types of effects (in addition to ADD and DELETE), and by supplying two predefined predicates.

Creation is handled by the NEW effect, and deletion by the REMOVE effect. Examples appear in *mkdir* and *delete*, respectively. When an entity is created, a name can be specified for it; if none is specified, one will be generated automatically. An operator library must define what types of entities it uses, including both dynamically created and statically existing entities. This is done with the entities declaration seen in Appendix A following the operator definitions but before the predicate definitions.

There are two predefined predicates relating to entities. *Exists* takes two arguments, the type of the entity and the entity name; this predicate automatically becomes true when an entity is created (by NEW) and automatically changes to false when it is deleted (by REMOVE). *Equal* takes three arguments (a type and two entity names of that type) and returns true if the two entities are the same. The negation of *equal* is commonly used when an operator has two parameters that are the same type and it is known that they must be bound to different entities, as for example in the *construct* operator.

The fact that new entities can be created by operators cannot be handled solely by introducing these new predicates and effects. It also affects the way in which preconditions, constraints, and subgoals should be evaluated. If an expression is evaluated too early, then objects which are created later will have been improperly omitted from consideration. This is discussed further in sections 2.2.2.3.2 and 2.2.2.3.3.

2.1.2.2 Expressions and Computed Predicates

Goals, preconditions, subgoals, and constraints are expressions composed of domain-state predicates combined with *and*, *or* and *not* operators. There are no quantifiers, but negation can be used creatively to compensate for the absence of quantifiers (examples are given below). This is because *not* is implemented in the PROLOG fashion: the expression *not f(x)*, where x is unbound, succeeds only if there is no entity for which the predicate f is true. As a result, care must be exercised in using *not* : evaluation of *not f(x)* will not identify all x's for which *not f* is true; if x is already bound, evaluation will return the expected answer.

All the predicates that appear in the original state schema shown in Figure 2.2 need to be recorded explicitly to capture the world state. However, many of the new predicates introduced in Figure 2.4 should not be explicitly recorded, but rather should be computed

from those that are explicitly recorded. These are called computed predicates. The SIPE facility of deductive operators partially addresses the same problem.

Consider a predicate like *sys-in-dir*. If this predicate is explicitly stored, then every single operator that affects what files are in what directories must update the predicate. This is dangerous because it is so easy to overlook a case; it is also complicated, requiring a facility for conditional effects (only if the file contains a part of some system is *sys-in-dir* affected). Instead, using a "macro" defining the meaning of *sys-in-dir*, the truth value can be determined in terms of predicates that are explicitly recorded. In the case of *sys-in-dir(s,d)*, this definition is that every part of s is represented in directory d. It uses a further computed predicate, *represented-in-dir(p,d)*; contents p is *represented-in-dir* d iff there is a file f whose contents are p and f is in d.

Expression of these definitions must overcome the obstacle of the absence of quantifiers (specifically the for-all quantifier) in the expression language. "Every part of s is represented in d" can also be expressed as "there is no part of s that is not represented in d". Thus, the use of double negatives in the computed predicate definitions is intentional. (Use of this stratagem is not limited to the computed predicate macros— it can be applied to any expression in an operator definition. However, more readable operator definitions result when these complex expressions are made into computed predicates so that only the predicate, not its convoluted definition, appears in the operator definitions. Also, some of the computed predicates have internal arguments, such as f in *represented-in-dir*; using computed predicates avoids having to make these into additional parameters in the operators.)

Computed predicates cannot appear in the effects clause of an operator; in fact, it is to avoid this that they were defined in the first place. When they appear in goal, precondition, subgoal or constraint expressions, the predicate definition is substituted at the time the

expressions are evaluated. Examples of computed predicate definitions, each giving a predicate with its parameters and macro expansion, can be found at the end of Appendix A.

2.1.2.3 Preconditions

Two types of preconditions are used in the example: *static* and *dynamic*. *Static* preconditions define conditions of applicability for operators; if a static precondition is not true, then the operator is not applicable and no effort should be expended to make it applicable by making the precondition true. For example, in the *delete* operator, there is a static precondition that the file to be deleted already exists. This precondition is static because there is no sense in creating a file just in order to delete it. A *dynamic* precondition is just a special kind of subgoal, distinguished by the fact that it must be achieved before any of the subgoals are achieved.

In practice, static preconditions can be used to ensure that the hierarchical structure of actions reflects the user's perspective on the domain. For example, the *move* operator has a static precondition that the directory into which the file is being moved already exists. In this case, the creation of directories is seen as part of a higher-level activity (here, committing reference and working directories). It just so happens that there is an ordering constraint between directory creation and moving files. The precondition on *move* cannot be omitted, or improper action sequences will be allowed. But, in attaching a *mkdir* action into a plan, we want to capture the fact that its primary "purpose" is part of *commit-refdir* or *commit-workdir*, not part of the *move*.

2.1.2.4 Constraints

Constraints are restrictions on parameter bindings in operators. For example, in the *construct* operator, there is a constraint that the working directory and the reference directory cannot be the same. The difference between static preconditions and constraints

has to do with the time at which the condition is checked. Preconditions must be true when an operator starts, and are checked then. All constraints must be true when an operator starts, but it is often possible to check constraints before this time (see section 2.2.2.2.3).

The distinction between constraints and static preconditions is a subtle one. As a matter of operator definition style, we use preconditions to specify restrictions on the ordering of actions and constraints to specify restrictions on parameter bindings. Thus in the *delete* operator, where the restriction that the file be in some directory is used to uniquely identify the *dir* parameter in the operator, the restriction is given in a constraint. not a static precondition.

2.1.2.5 Iterated Subgoals

Several operators use the concept of iterated subgoals, where a single subgoal expression is achieved for one or more parameter bindings. Two different types of iteration are provided. In one form, iteration occurs until a particular logical expression E becomes true; in the other, a value N is specified such that the subgoal must be achieved at least N times. SIPE takes a slightly different approach to iteration, using a generator function to indicate the appropriate set of parameter bindings.

Iteration *until(E)* is used in both *place-parts* and *remove-extra-files*. The subgoal that is achieved repeatedly involves placing a file either into or out of a particular directory. The expression E is one of *sys-in-dir* or *only-sys-in-dir*. In *place-parts*, for example, different files in different source directories are involved in the different iterations; but all iterations involve moving into the same target directory. Note that in both *place-parts* and *remove-extra-files*, E becomes true as a side-effect of the final iteration. This need not be the case; generally, the operator that achieves E need not be one that also achieves one of the iterations.

Iteration for N=1 is used in *make-components*. There, we require that a system have at least one new part, leaving unspecified the exact number that are required.

When iteration is *until(E)*, completion of the iterated subgoal can be determined by testing whether E is true. When iteration is via a minimum count, there are no corresponding completion criteria. Anytime after N iterations, the operator could be done. As a practical matter, such operators are only considered completed when completion is implied by later activity; for example, if the iterated operator is achieving a precondition of another operator, then completion is implied when that other operator starts. Details on implementing this form of iteration completion appear in section 2.2.2.5.

2.1.2.6 Offline Operators

Not all actions in a process are necessarily observable. For example, consider the various ways that *define_system* might be carried out if we were to model that activity in greater detail. The programmer might edit a text document that is a (formal or informal) specification of the system, there might be mail exchanged between the programmer and the customer about the requirements, or there might be a conversation to discuss the requirements. In the first two cases, there are overt actions (editing and sending/receiving mail) to observe, but in the last case there is no "on-line" action to observe. The best that can be done is to infer, when other actions occur that imply that *define_system* has finished, that a requirements discussion must have taken place. Other examples of such "offline" actions include learning about bugs through conversation, passing a design/code review, decisions made at a project or configuration control board meeting, and the like.

There is also another type of unobservable action: some decision-making by the programmer may be significant enough to be represented as a separate "action" in its own right. In the setup-environment example, there is one key decision that is largely hidden amongst the actions: the decision about which baseline to use in building the new system

version. The use of a separate operator to represent choosing the baseline leads to better modeling of the process, especially in the case where there are other actions that could be taken as a precondition to making the baseline choice. For example, the programmer might examine (edit without change) some of the source code in a potential baseline as part of making this decision. Having a separate operator for choosing the baseline helps to properly represent the role of these supporting actions. (See section 3.2.3 for a further development of this.)

Offline actions are defined just like any other primitive operator, except that they have the keyword "offline". The goal of an offline action may be the same as the goal of regular actions (as in the *define-system* example described above), or it may be unique (as in the *choose-baseline* example). A simple version of the offline operator for *choose-baseline*, and the revised version of *setup-refdir* that goes with it, are given in Figure 2.7. The recognition of offline actions is described in section 2.2.2.5. The use of the offline operator *choose-baseline* is further elaborated in section 3.2.3.

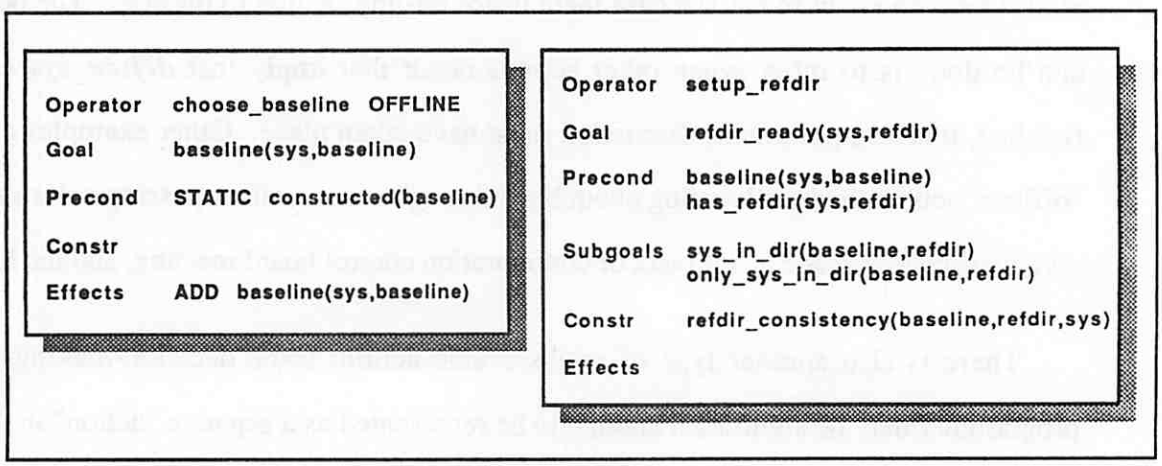

Figure 2.7 Example of Offline Operator

2.1.2.7 Other Features

There are a variety of other features, useful in complex domains, that can be added to an operator definition language as an engineering activity. We give several examples here; they are discussed in more detail in [Huff & Lesser, 1987].

Conditional effects allow a single operator to describe the outcome of an action that varies slightly with the state of the world in which it is executed; without conditional effects, multiple operators with different preconditions must be used. (Planning with conditional effects is developed in [Pednault, 1988].) If the planner/plan recognizer is to "optimize" plans, it is necessary to have information about the relative cost of operators, so that least cost alternatives can be used where possible. Costs may be measured on one or more types of resources (in computer-based domains, CPU time, I/O time, human time, disk space, etc.). The subgoal decomposition of complex operators can be elaborated so that an operator can specify additional ordering restrictions among subgoals or restrict the achievement of some subgoal to certain operators. However, it must be noted that (if treated as requirements rather than advice) this additional information is an escape mechanism—there is no "understanding" of these restrictions. They simply allow the designer of operators to say "Trust me, this is how it is."

2.2 Plan Recognition

Plan recognition is the process by which a plan, and its goal, are inferred given an initial state and a sequence of actions. In the remainder of this chapter, we present a basic plan recognition algorithm that is suitable for the intelligent assistant application and that supports the operator definition language just described. This algorithm is implemented in the GRAPPLE system. The pseudocode for this algorithm appears in Appendix B while examples of the execution of this algorithm appear in Appendix C.

2.2.1 Overview of Plan Recognition

Plan recognition has proved a useful paradigm for a variety of applications, such as automated consultation [Genesereth, 1979], natural language discourse [Allen, 1983; Sidner & Israel, 1981], story understanding [Wilensky, 1981], explanation-based learning [DeJong & Mooney, 1986], and interpreting behavior of adversaries [Azarewicz et al., 1986]. Applications tend to differ in the assumptions that can be made about the action sequence to be recognized, and plan recognition algorithms differ accordingly. An algorithm suitable for the requirements of story understanding, based on the idea of marker passing, is described in [Charniak, 1986]. Another algorithm, emphasizing the ability to draw conclusions about the present and about future actions in spite of incomplete or out-of-order action information, is described in [Kautz, 1987]. A system for plan recognition that is not based directly on the classical planning formalism is described in [Azarewicz et al., 1986].

We first consider the requirements of the intelligent assistant application, and then discuss the role of search in plan recognition.

2.2.1.1 Requirements of Intelligent Assistance

In some applications, it is appropriate to make three assumptions: that the action sequence is complete and includes only primitive actions, that the order of the actions is known exactly, and that there is exactly one top level goal. Such assumptions may be appropriate in automated consultation, for example [Genesereth, 1979]. At the opposite extreme, none of these assumptions is adopted, implying that actions may be missing or represented only at a higher level of abstraction, that the exact order of the actions may not be known, and/or that multiple top level goals may be in progress simultaneously. These conditions arise in story understanding and some natural language applications. In some applications (story understanding), it can be assumed that all actions are legal, while in

other applications (automated consultation), finding the "illegal" action(s) is a primary objective.

The plan recognition requirements of intelligent assistance are as follows. It can be assumed that only primitive actions are observed, that all the primitive actions (except offline actions) that have occurred up to the current point in time are known, and that their exact order of occurrence is known, but that the action sequence may well be incomplete; that is, future actions, as yet unknown, may be required to finish a plan that has been started. In addition, there may be more than one top level goal being achieved at any given time, and consequently more than one plan in progress. And finally, some of the actions may be invoked (by the user) in error; identifying these errors, before the action is actually executed, is an important service in intelligent assistance.

Since we assume that the observed actions are always primitive operators, we will henceforward use the term action to mean an instance of a primitive operator.

In the plan recognition algorithm described below, it is assumed that the operator library is complete, i.e., that all actions can be explained by the given operators. In intelligent assistance, when this assumption is made, action sequences that do not fit the given operators are reported as errors. It is desirable to relax this assumption, and to provide some facility for additional analysis to explain unconventional action sequences. One system which does this is described in [Broverman & Croft, 1987]; there, unexpected occurrences are actually used to augment the operator library dynamically. A plan recognition system that generates partial interpretations when the operator library is incomplete is described in [Wills, 1987].

2.2.1.2 The Role of Search

The assumptions about the action sequence, and the nature of intelligent assistance where the actions occur incrementally, suggest a cyclic algorithm—where each cycle consists of getting the next action and then interpreting that action. (If the entire action sequence were available, then a multi-pass approach to recognition would be possible.) In general there will be multiple competing interpretations for the actions that have occurred to date. Each interpretation represents one possible view of the purposes of and the interrelationships among all the actions. An interpretation has two parts (both as defined in Chapter 1): a plan network and the world state corresponding to that plan network (see, for example, Figure 1.7). The plan network shows one or more plans in progress, and how each action contributes to the completion of these plans. The world state represents the effects of all the actions (primitive operators) taken and all complex operators that are known to have completed.

(Note that two competing interpretations covering the same actions can have different world states. For example if one interpretation of *mkdir* D is that it is part of *commit-refdir* for system S, then *has-refdir(D,S)* will be true in the world state for this interpretation; if there is a competing interpretation that *mkdir* D is part of *commit-workdir* for system S, then *has-workdir(D,S)* will be true in its world state.)

Plan recognition is fundamentally a search problem, where the search is conducted through a tree of interpretations, as indicated in Figure 2.8. The single interpretation at the top (level 0) corresponds to the initial world state and an empty plan network. Figure 2.8 shows that there are four ways to interpret action 1, given on level 1. After action 2 occurs, there are six possible interpretations, shown on level 2, that account for both actions 1 and 2. Note that action 2 is inconsistent with one of the interpretations accounting for action 1 alone (this is shown with the "X" in the figure.) That is, this interpretation

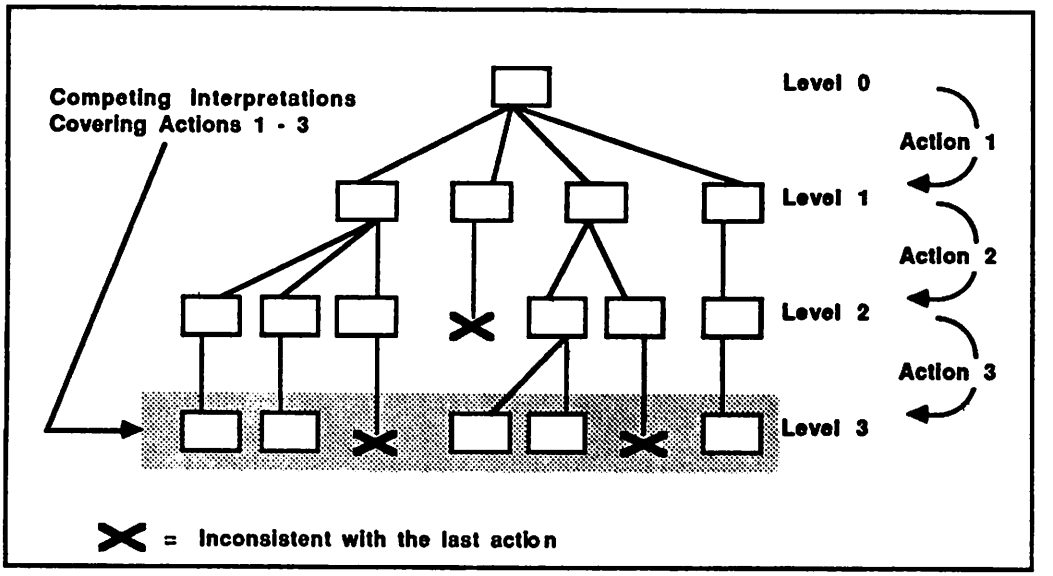

Figure 2.8 Tree of Interpretations

looks viable given only the information that action 1 occurred; however, since it cannot be extended to account for action 2 as well, it has to be abandoned as a possibility.

Each cycle (i.e., each successive action) adds another level of expansion in this tree of interpretations. Interpretations at level N in the tree account for the first N actions; they compete with one another. An interpretation at level N extends its parent at level N-1 by adding a rationale for the N^{th} action. When action N is recognized, some of the interpretations at level N-1 cannot be extended to incorporate a rationale for the new action (these are marked with an "X"); some can be extended in a single way; and some will support multiple extensions; each of these situations occurs in Figure 2.8.

Thus, a key issue in plan recognition relates to the traditional concerns in search: pruning alternatives as soon as possible, identifying the most promising paths to pursue, and avoiding expending effort on the least promising paths. Following all alternatives is simply too expensive. In the plan recognition algorithm described below, the search is

controlled in two ways: through aggressive use of the knowledge already contained in the operator definitions to prune alternatives quickly, and through the use of additional knowledge in the form of heuristics to identify the "best" alternative to pursue. These are discussed in detail in sections 2.2.2.2 and 2.2.2.3 respectively.

Control of search is especially important in a recognizer for hierarchical plans, where interpretations proliferate due to parameter behavior. In a given complex operator, each subgoal will not necessarily mention all the parameters of the operator; different subgoals will use different subsets of the parameters. Thus, the actions that are taken at the beginning of a plan will not in general bind all the parameters in the plan. The earlier bindings can be found for parameters, the earlier interpretations can be ruled out, by identifying incompatibilities between bindings in existing plans and bindings on the action being interpreted (this is discussed in more detail in section 2.2.2.2.1). Thus, the earlier parameters can be bound throughout a plan, the fewer spurious interpretations there will be for subsequent actions. We will show that aggressive use of the knowledge already contained in the operator definitions can provide ways of bindings parameters, so that the plan recognizer does not have to rely solely on actions to bind parameters.

2.2.2 Plan Recognition Algorithm

In the following sections, we describe a plan recognition algorithm suitable for the intelligent assistance application. An overview of the plan recognition architecture, showing both algorithm and data structures, is given in Figure 2.9. The algorithm has four major parts: identifying candidate interpretations, pruning, focusing, and updating status; the role of planning in plan recognition is discussed as an additional topic. The pseudocode for the algorithm is given in Appendix B.

The depiction of plan networks used in discussing the algorithm differs somewhat from that used in Chapter 1 (as in Figures 1.5 and 1.7), and the translation between the two

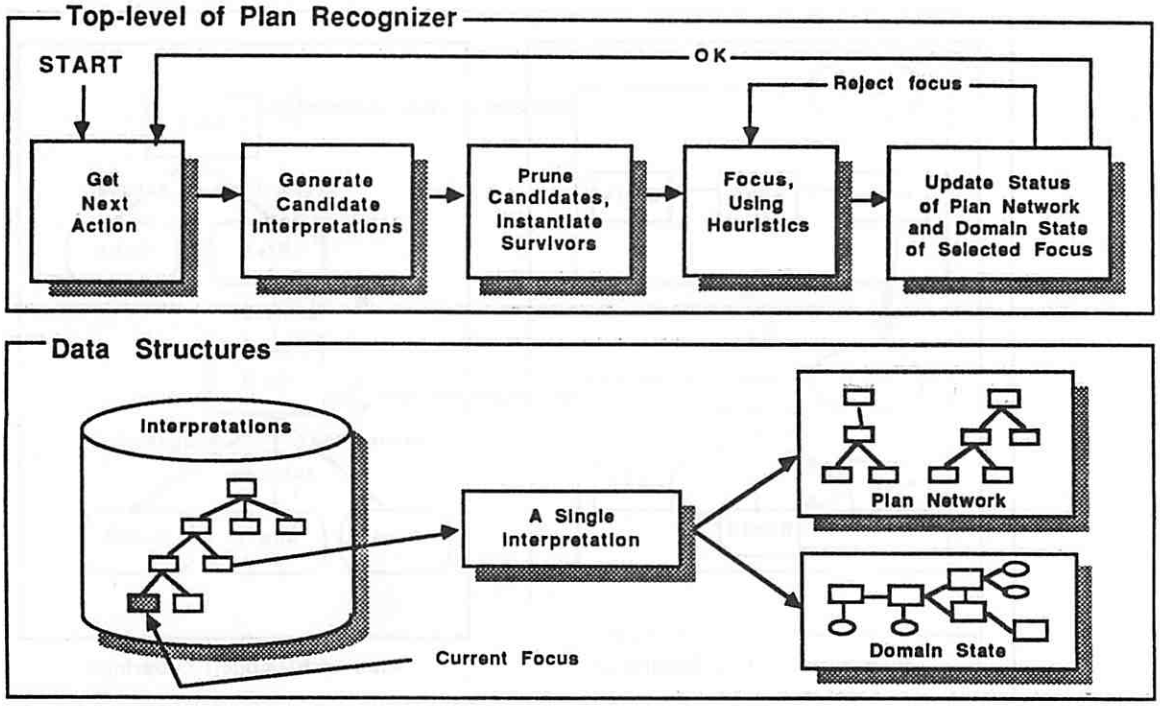

Figure 2.9 Architecture for Plan Recognition

is shown in Figure 2.10. The new depiction emphasizes the hierarchical levels of the plan network, while the old emphasizes temporal constraints within a level.

2.2.2.1 Identifying Candidate Interpretations

An explanation for action N is an extension to a specific interpretation covering actions 1 through N-1, resulting in an interpretation that covers actions 1 through N. The extension is dependent both on the plan network and the world state in the interpretation for the preceding actions. There are two types of extensions. An extension may continue a plan already started (to the left in Figure 2.11), in which case it consists of a path from the action (which is a primitive operator) up to a pending precondition/subgoal already manifest in the plan network. Alternatively, the extension may start a new top level plan (to the right

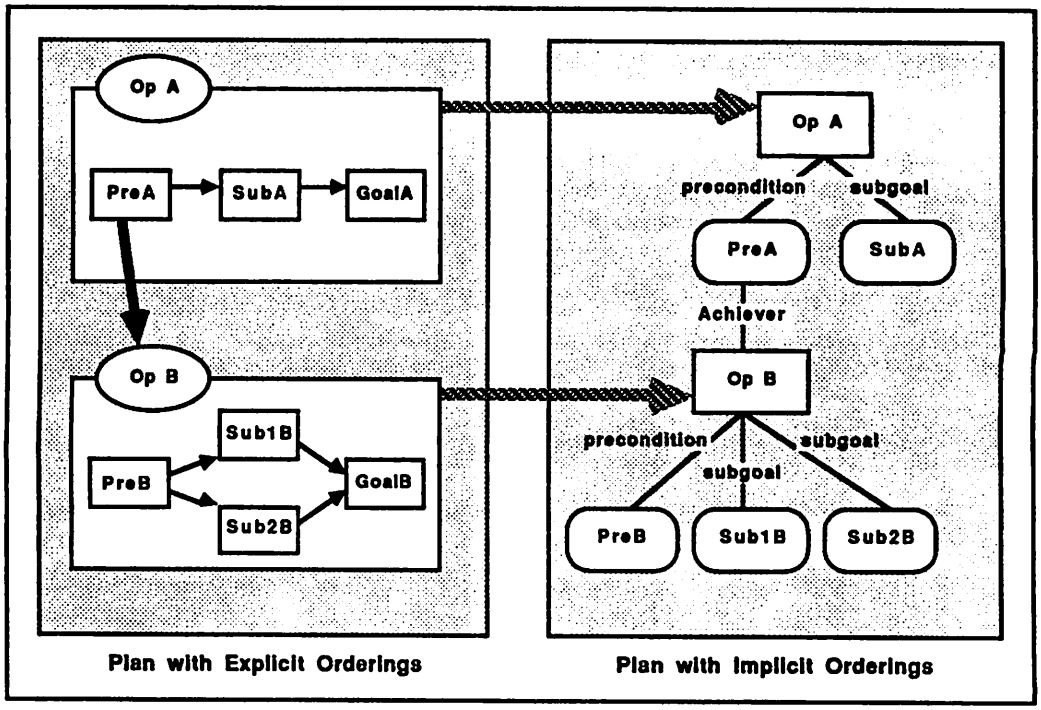

Figure 2.10 Two Plan Network Depictions

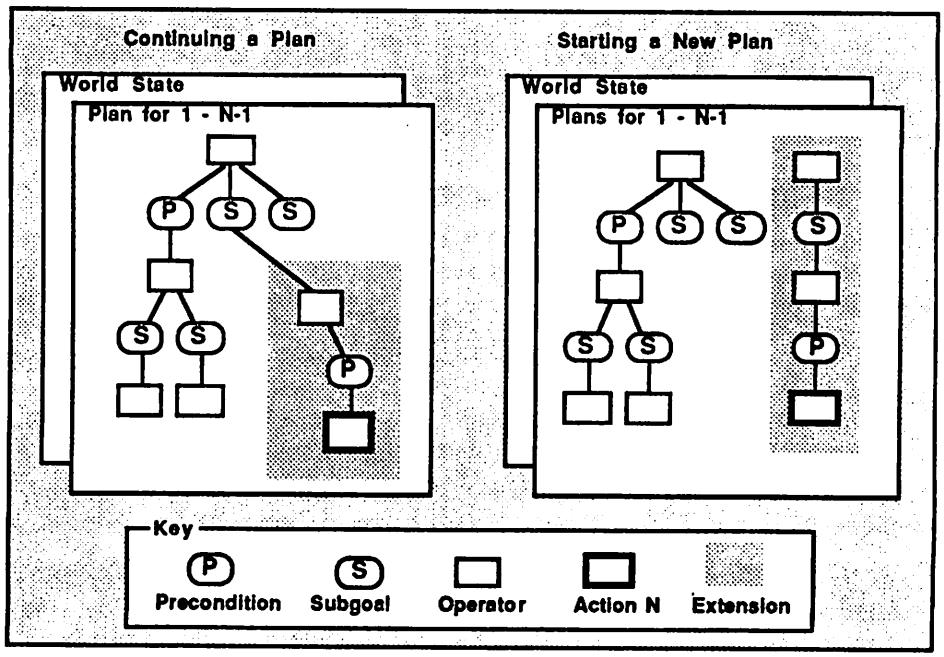

Figure 2.11 Two Types of Extensions to Interpretations

in Figure 2.11), in which case it consists of a path from the action to a precondition/subgoal in a new instance of a top level operator.

2.2.2.1.1 Paths in Plans

Given an interpretation for actions 1 through N-1, there is a possible extension for every path from action N to a pending condition in the interpretation or to a top level operator new to the interpretation. Each of these extensions leads to a new interpretation, and all these potential interpretations must be considered, subject to further viability checks; typically some will be discarded for failing these checks.

The path that defines an extension consists of one or more steps that connect an operator O1 with the subgoal or precondition of another operator O2, where the goal of O1 "achieves" the subgoal/precondition of O2. Each such step takes the path one step up in the hierarchical view of actions. The path can be thought of as the rationale or purpose of the action, and it captures the relationship of this action to the previous actions (or some subset thereof). The legal steps that can appear in a path can be analyzed in advance, and this analysis is described in the next section.

2.2.2.1.2 Analysis of Achievers

Each operator (except the top level operator) appearing in a plan has a "purpose": to satisfy a subgoal or a precondition of another operator. (Recall that static preconditions are never to be explicitly achieved, so we restrict our attention here to dynamic preconditions only.) Although potential relationships between operators and the preconditions or subgoals they could achieve are considered during design of the operator library, these relationships are not explicitly given in the operator definitions. (These are the *achiever* relationships shown in Figure 2.6). There is enough information in the operator definitions

to deduce what operators can achieve what preconditions and subgoals, and this inference process can be performed in advance and saved, before any plan recognition is started.

Let the set of dynamic preconditions and subgoals that appear in each operator in the operator library be called the dynamic *states*. An operator achieves a dynamic state if the goal of the operator implies that the state holds. Obviously state A (where A stands for some expression composed of domain predicates with arguments) is achieved by an operator whose goal is A. But exact matches are not the only cases of interest. For example, state A is also achieved by an operator whose goal is A AND B.

Formally, achievability is decidable through resolution refutation. Given a state, consider each operator (actually its goal) in turn. If a contradiction can be derived from NOT (IF <goal> THEN <state>) taken together with the definitions of the computed predicates, then the operator is an achiever of the state. For example, *copy* achieves the subgoal *moved-to-dir* in *place-parts*, because the goal of *copy* exactly matches the expression defining this subgoal. *Move* also satisfies this subgoal; in this case, the goal of *move* actually "over-achieves" the expression. These relationships can be seen in Figure 2.6.

Note that when operator A achieves state B by this definition, this does not imply that operator A can always be used to achieve state B. A is only applicable when its preconditions and constraints are satisfied in the dynamic situation in which it is being considered for application.

At the same time the achievers are computed, a parameter map can also be derived. If the dynamic state is *f(x,y)* and the operator goal is *f(a,b)*, then the parameter map is <(x,a), (y,b)>. Only variables that appear in the state need be mapped, but all mapping possibilities must be considered. If the state is *f(x,y)* and the goal is *f(a,b) and g(a,c)*, then

the map does not mention c. If the state is *f(x,y)* and the goal is *f(a,b) and f(c,d)*, then there are two maps: <(x,a),(y,b)> and <(x,c),(y,d)>.

Achievers represent single-step reachability (adjacency) from an operator O1 to the precondition/subgoal C1 of another operator O2. This information can be stored in a relationship adjacent(O1, C1, O2, map). From this, n-step reachability (for n>0) can be computed; it can be stored in a relationship showing that operator A has a path to state B of operator C where a possible first step on the path is to state D of operator E. Using this reachability information, one can immediately determine if a given operator (representing a new action) has a path to a pending precondition or subgoal in the plan network, or to a new top level plan. And, one can generate all such paths, step-by-step.

The adjacency information for the operator library of Appendix A has already been shown in Figure 2.6 as the achiever links. All paths from primitive operators to top level operators can also be seen in Figure 2.6 by reading up from the primitive operators to the top. When an extension is proposed, it either contains one of these paths in its entirety (in the case that the action starts a new top level operator) or some initial part of one of these paths (in the case that the action continues an operator to which prior actions have already contributed.)

The achievers calculation presented here is not completely general; it makes several simplifying assumptions. If a single state has the form *A AND B*, the fact that achievers for B alone are sufficient when A is already dynamically achieved (and vice versa) is not considered. And, if the effects of operator A include ADD X but X is not mentioned in the goal of A, then A will not be regarded as an achiever for a state X. These simplifications are not burdensome in writing operator libraries. For example, to handle the case where A or B can be achieved separately, two subgoals (one for A and one for B) are used in lieu of a single subgoal for *A AND B;* this ensures that the granularity of the subgoals matches the

granularity of the operators. On the other hand, there is no theoretical problem with lifting such restrictions.

2.2.2.1.3 Static and Dynamic Namescopes

Each *operator definition* represents a separate *static* namescope. That is, the use of parameters within operator A is completely independent of the use of the same or different parameters within operator B. The parameter map that is computed along with the *achiever* relationship represents a mapping of parameter name between two static namescopes. For example, *place-parts* can achieve the subgoal *baseline-in-refdir* of *setup-refdir*; the parameter map indicates the parameter correspondences that hold when an instance of *place-parts* is used to achieve that particular subgoal in *setup-refdir*; other parameter correspondences hold for other uses of *place-parts*. The map in this case defines the correspondence between *sys* in *place-parts* and *baseline* in *setup-refdir*, and between *dir* in *place-parts* and *refdir* in *setup-refdir*.

Each *plan* represents a separate *dynamic* namescope. That is, the binding of parameters within plan A are completely independent of the bindings of the parameters within plan B, even if plan B uses some/all of the same operators as in plan A. A plan consists of instances of operators, each with a separate static namescope and each fulfilling a role that determines how names are mapped pairwise between these static namescopes. The purpose of the dynamic namescope is to provide a single set of names that can be bound to domain entities. There will be one dynamic name for each unique parameter in the plan; two different dynamic names may or may not be bound to the same domain entity. If there is a map for each operator instance translating its static names to this set of dynamic names, then the use of different names in different static namescopes to refer to the same parameter is resolved across the entire plan.

Figure 2.12 shows a plan and its dynamic namescope. Only a part of one path in the plan is given, involving three operator instances (and therefore three separate static namescopes). One static to static namemap is shown, relating the names in *place-parts* to the names in *setup-refdir* through the subgoal *baseline-in-refdir*. If the correspondences between the static namescope of *setup-refdir* and the dynamic namescope have already been determined, then this static to static map can be used to determine the correspondence

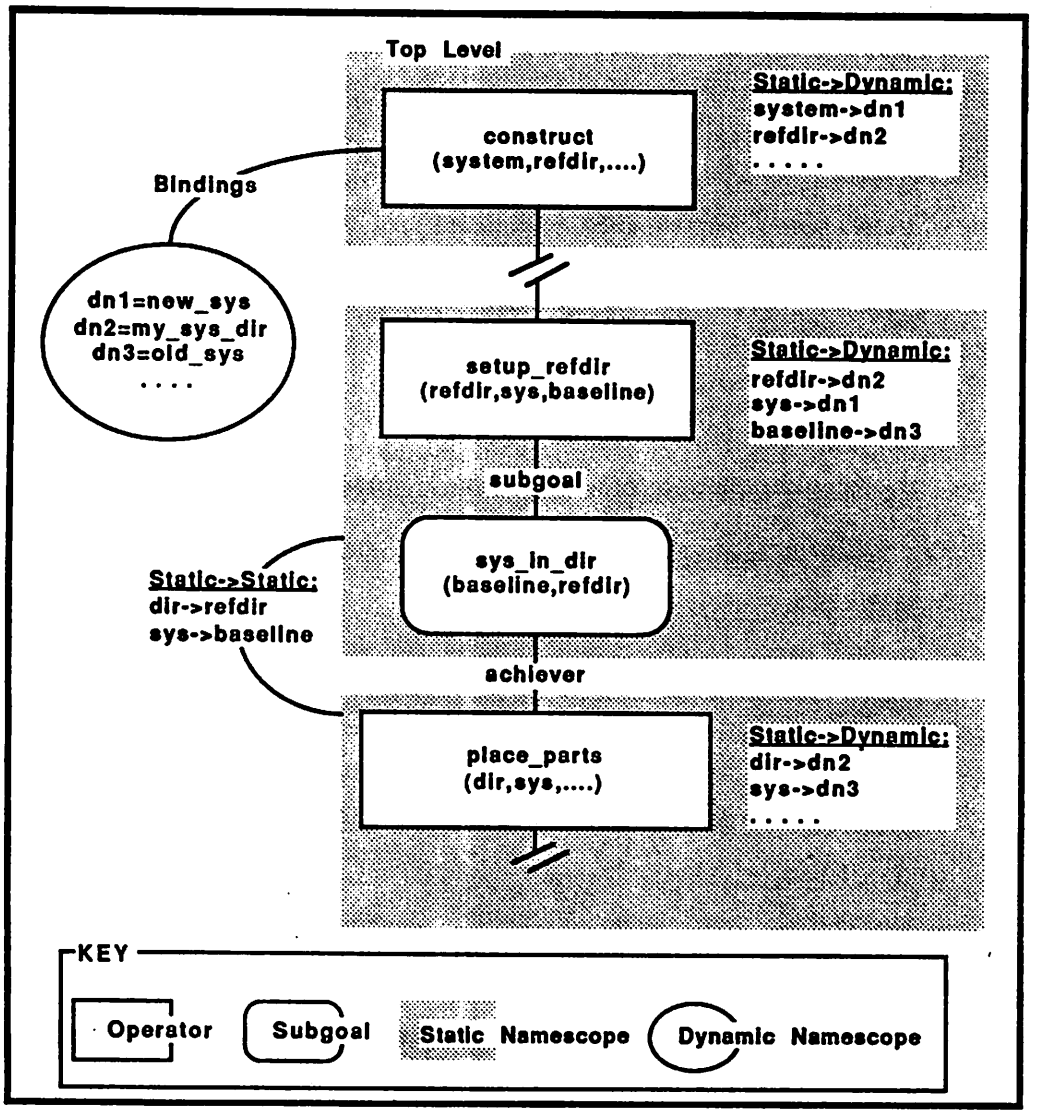

Figure 2.12 Static and Dynamic Namescopes

between the static namescope of *place-parts* and the dynamic namescope. This leads to the static to dynamic map given for *place-parts*. Using the static to dynamic map, and any bindings for the dynamic names that may already be known, any expression appearing as the goal, precondition, etc. in any operator instance can be evaluated. For example, the goal of *place-parts* is *sys-in-dir(sys,dir)*; this translates to *sys-in-dir(dn3,dn2)* in the dynamic namescope, and with the bindings shown becomes *sys-in-dir(old-sys,my-sys-dir)*.

The computation of static to dynamic namemaps is done at the time a path is formed, giving one new namemap for each newly instantiated operator that appears on the path. Without these namemaps, the expressions appearing in the operator cannot be evaluated. Some of these expressions must be evaluated to complete the validity checks on candidate interpretations, as described in the next section.

2.2.2.2 Pruning Interpretations

A possible interpretation for actions 1 through N must meet a collection of preliminary viability tests, which are described in this section. If these tests are not met, then the interpretation can be pruned (i.e., discarded) before it is even instantiated. If these tests are met, then the interpretation can be instantiated in anticipation that it will prove to be valid. Some of these instantiated interpretations may later be discarded after the effects of this action have been posted and the plan network status updated, or after later actions are recognized.

Identifying potential interpretations, by finding possible path extensions, is a process of finding interpretations that are legal *syntactically*. Applying the viability tests described here is a process of restricting attention to those that are legal *semantically*. In a typical parsing process, the entire input is first checked syntactically, and then (perhaps in multiple passes) checked semantically. Since the entire action sequence is not available during plan

recognition, this multi-pass approach is not feasible, and the semantic checks are therefore interleaved with the syntactic ones. One further difference should also be mentioned. In parsing, considerable effort goes into expressing grammars in such a way as to reduce the look-ahead required to resolve ambiguities over which grammar rule to apply; the theoretical results that make this possible apply to context-free grammars. The syntactic structure implicit in operator definitions allows interleaving, and is not based on nested closure; it is NOT context-free. (An alternative approach, based on viewing a plan grammar as a *graph grammar* that is context-free, is given in [Wills, 1986].)

In addition to pruning certain interpretations, application of these tests serves to bind additional parameters in the plan network. Prompt binding of parameters helps to prune potential interpretations for later actions.

The checks are described one-by-one below; the ordering of the checks is as follows. The parameter consistency check is made first. Then the other checks are applied by following the path from the action all the way to the top level operator. (In the case that the path leads to an existing pending condition, the top is reached by traversing the existing path from that pending condition to the top level operator). At each operator on the path, the precondition, constraint, and goal tests are applied. Finally, after the path is traversed, the goal duplication check is applied. During this process, the interpretation can be rejected at the first failure of a viability test. This order of checking is the approach to pruning and binding which is implemented in GRAPPLE and therefore applied to the test cases shown in Appendix C.

2.2.2.2.1 Parameter Consistency

When an action continues a plan already in progress, the parameter bindings given with the action being recognized must be consistent with any bindings that have already been determined for that plan. For example, suppose there is an existing plan that includes

a subgoal *refdir-ready(S2,X)* being achieved by the operator *setup-refdir* which has a pending subgoal *sys-in-dir*. In this existing plan, the parameters of *setup-refdir* are bound as follows: refdir=X, system=S2. Let the action now being interpreted be *move* where from=F, to=NEWF, to-dir=Z. If this *move* satisfies the subgoal *sys-in-dir* of *setup-refdir*, then (by the static-static parameter map as discussed in section 2.2.2.1.3) *to-dir* of *move* is equivalent to the *refdir* of *setup-refdir*. But, this cannot be because Z and X are different directories. Therefore, this instance of *move* fails the parameter consistency test, at least with respect to achieving the pending subgoal *sys-in-dir* in this plan. If the *move* had involved X as the target directory, then the parameter consistency test would be met and there would be a legal path from *move* to *sys-in-dir*.

This test is irrelevant in the case of an interpretation that starts a new plan—there are no predetermined bindings. Instead, the bindings of the action are used to initialize the bindings of the new plan.

2.2.2.2.2 Precondition Satisfaction

Precondition satisfaction must be considered along the entire path from the action to the top (including the new path part and the existing path part if any). Whether the precondition must actually be true now depends on the current state of the operator and whether the path is through a precondition or subgoal to that operator. All operators in the plan network have a status, which is one of waiting-preconditions, preconditions-satisfied, started, or completed. (Note that preconditions are not required to stay true once an operator has started.) All operators on the new part of the path have an implied status of waiting-preconditions.

Consider operator O on the path. If O is primitive (i.e., if O is the current action), then its preconditions should be true now, and must be tested. If O is complex, and the path to O is through one of the preconditions of O, then obviously the preconditions do not need to

be true—this action is attempting to help satisfy those preconditions. In this case there is no precondition test applied. If O is complex and the path to O is through one of the subgoals of O, and if O has a status other than started, then O is implicitly starting now and its preconditions must be true and should be tested. If O is complex and the path to O is through one of the subgoals of O, and if O has a status of started, then O has already started and its preconditions are no longer required to be true. In this case there is no precondition test applied. Note that if O is on the path, it cannot have a status of completed because it would have no pending precondition or subgoal for the action to satisfy.

Testing preconditions ensures that attention is restricted to those achievers that are actually applicable in the current dynamic situation; this compensates for the fact that the achievers calculation ignores dynamic factors. At a minimum, the preconditions of the action itself must be true in the current world state. Thus it is an error to move a file to a directory that does not exist; no interpretation can be made for such an action because all interpretations fail the preconditions test. For a complex operator, the preconditions of that operator must be true at the time the first action is taken whose interpretation involves a subgoal of that operator. Thus, it is not proper to perform any action to satisfy the subgoals of *setup-refdir* (which involve placing files) until the precondition is satisfied (i.e., the directory has actually been committed for this use).

2.2.2.2.3 Constraint Satisfaction

Two different constraint satisfaction tests are applied: one for operators that are starting and one for operators that are still waiting for their preconditions to be satisfied. The distinction is necessary because of the fact that new entities can be created in the world state by some of the operators. It allows constraints to be checked earlier than is otherwise possible, which helps to prune interpretations earlier.

When an operator starts, all its constraints must be met. Thus for a path from *move* to *place-parts* to *setup-refdir* to be viable, the contents of the file being moved must be part of the baseline the reference directory is intended to hold. And for a path from *move* to *remove-extra-files* to *setup-refdir* to be viable, the file being moved must be in the reference directory.

Sometimes constraints can be checked before the operator they appear in has started. To apply the constraint check early, one must have bindings for all the parameters appearing in the constraint. For example, if we are considering a path leading to the precondition *env-setup* in *construct*, and we have bindings for both the *refdir* and the *workdir*, then we can check to see that they are not equal. However, if we know only the *refdir*, we should not evaluate the constraint—to do so would restrict the *workdir* to those directories that now exist (excluding of course the *refdir*), which in turn means that a perfectly legal future action of *mkdir* to create a (new) directory to be the working directory would be flagged as an error.

2.2.2.2.4 Goal Not Achieved

It is clear, but perhaps not obvious, that the goal of an operator cannot be true if the operator appears on the path being checked. Actions should not be taken to achieve something which has already been achieved and is still true. However, the test for "goal not achieved" can only be applied when there are already bindings for all parameters in the goal; otherwise the test is overly restrictive in the presence of operators that create new world entities (this is the same phenomenon as described for constraint satisfaction).

This check helps to bind parameters and/or to prune interpretations. For example, if there are several existing systems only one of which (S1) is not yet constructed, and the proposed action is a *mkdir*, then all the competing interpretations involve some path to a subgoal of the *setup-env* operator and thence to the (single) precondition of *construct*.

Assuming *setup-env* has not yet started, the precondition of *setup-env* will be tested (as described in section 2.2.2.2.2); this limits binding of the parameter *system* (in both *setup-env* and *construct*) to the set of existing systems. But, *constructed(system)*, the goal of *construct*, is already true for every system but S1; therefore applying the goal-not-achieved check serves to bind the parameter *system* uniquely to S1. If all existing systems were already constructed, all the interpretations would fail the goal-not-achieved test, and they would all be pruned.

This test has the side benefit of saving the writer of operators from explicitly including <goal-not-true> as one of the operator's preconditions. For example, it is meaningless to do a *mkdir* if the directory named already exists, or to define a system which already exists. Since plan recognition has the test for "goal not achieved" builtin, as described in this section, these conditions do not have to be provided explicitly as static preconditions.

2.2.2.2.5 Non-duplication of Top Level Goal

There should not be two plans in progress simultaneously to achieve the same goal *for the same parameter bindings*. Thus, if there is a plan in-progress whose top level operator is *construct* with some binding for the system, then if the path under consideration leads to another *construct* operator with the same binding, we can prune the interpretation. Note that the same-bindings test is applied only to the parameters that appear in the goal expression.

This test is designed to prune spurious interpretations that arise due to the fact that multiple top-level goals are allowed. These spurious interpretations, involving starting a new top level plan, arise when operators have multiple subgoals that are not order dependent. Consider such an operator O, and suppose that one plan using O is already in progress with one of the subgoals S1 of O satisfied by action A1, as diagrammed on the left in Figure 2.13. A new action A2 that can be interpreted with a path to another subgoal

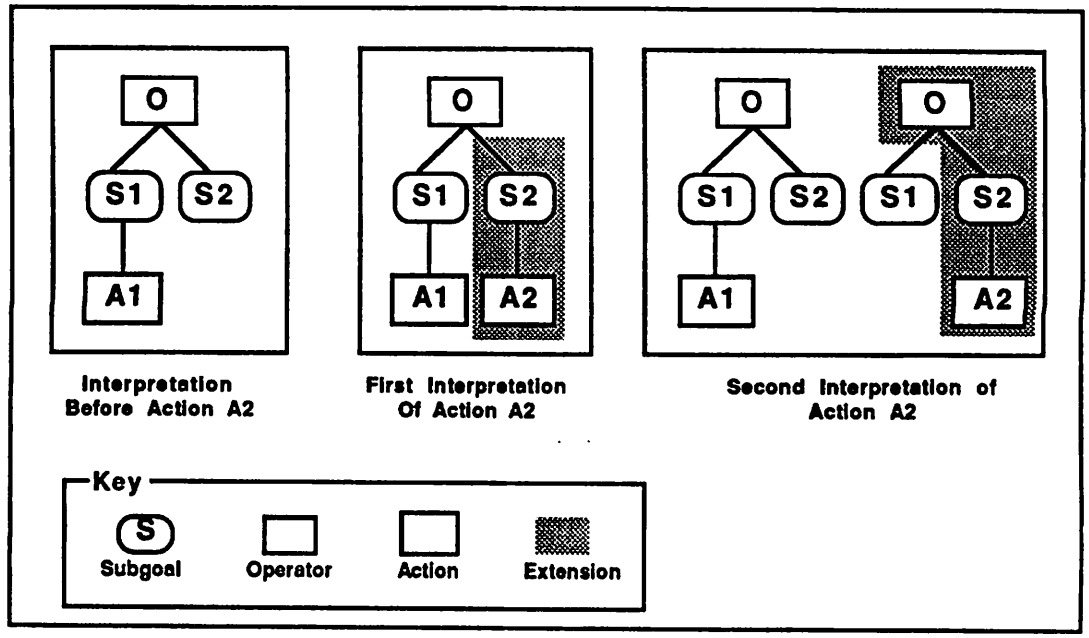

Figure 2.13 Proliferation of Interpretations

S2 of O will also have an interpretation that has a path through new instances of S2 and O in a new plan (assuming either that O has no precondition or that its precondition is true for some set of bindings). Thus, in this second interpretation (shown at the far right) there will be two instances of plans with O at the top level, one in which subgoal S1 is satisfied and one in which subgoal S2 is satisfied. If all parameters in the goal of O are bound in both instances, and if the bindings are the same, then the second interpretation can be discarded—essentially it splits actions among two plans which are equivalent, and the actions are properly accounted for in a single plan in the other interpretation (shown in the middle of the figure).

This test can be made more sophisticated to discriminate more accurately. As described, it cannot handle a situation where the original plan to achieve some goal has been abandoned before completion, and a new way of achieving the same goal started,

using different operators or using different bindings at lower levels within the plan. (This kind of scenario does not arise in all types of plan recognition, but it certainly does in the intelligent assistance application.) Similarly, more sophistication is needed to recognize that an interpretation of an action that starts a new plan for *construct(x)* with x bound to S1 or S2 is not viable when there are already two top level plans in the plan network, one for *construct(S1)* and one for *construct(S2);* here the new plan is subsumed only if both in-progress plans are considered. The simple form of the test described in the previous paragraph is currently implemented in GRAPPLE.

2.2.2.3 Focusing

A decision to focus on one instantiated interpretation in preference to its competitors can be made after instantiating the new viable interpretations. In general, several interpretations will pass all the syntactic and semantic tests described in previous sections. Additional knowledge must be used to differentiate the competing interpretations. Unlike the tests already applied, this knowledge is of a heuristic nature—it cannot be used to rule out possibilities, but rather to identify possibilities of higher likelihood. Some heuristics are suggested by the intelligent assistant application, and others are dependent on the specific domain (i.e., the specific operator library). An approach to implementing these heuristics, using truth maintenance systems, is given in [Carver et al., 1984].

Domain-independent heuristics appropriate in intelligent assistance are derived from the fact that there is usually continuity in successive actions. An action is more likely to be continuing a plan that is already in-progress than starting a new plan; also, continuing the plan of the previous action is more likely than switching to any other in-progress plan. These are heuristics, not absolute requirements—it does happen that people start new activities before finishing old ones, and that actions contributing to two separate activities are interleaved.

Domain-dependent heuristics distinguish between alternatives by their relative likelihood. For example, if operator A is the preferred method of achieving condition B of operator C, then an interpretation whose path involved A to B would be preferred to an interpretation whose path involved some other achiever for B.

The current implementation of GRAPPLE, and thus the examples presented in Appendix C, are based on a small set of heuristics. First, there is a preference for interpretations that continue in-progress plans to interpretations that involve starting new plans. Second, there is a preference for interpretations that do not require planning (as will be described in section 2.2.2.5).

By focusing on one interpretation now, it is possible to avoid both updating and expanding its competitors. If these focusing decisions are not later found to be wrong, then search will proceed as shown in Figure 2.14; only that part of the search tree that is shaded will ever be instantiated. Thus, the non-shaded part of the search tree represents

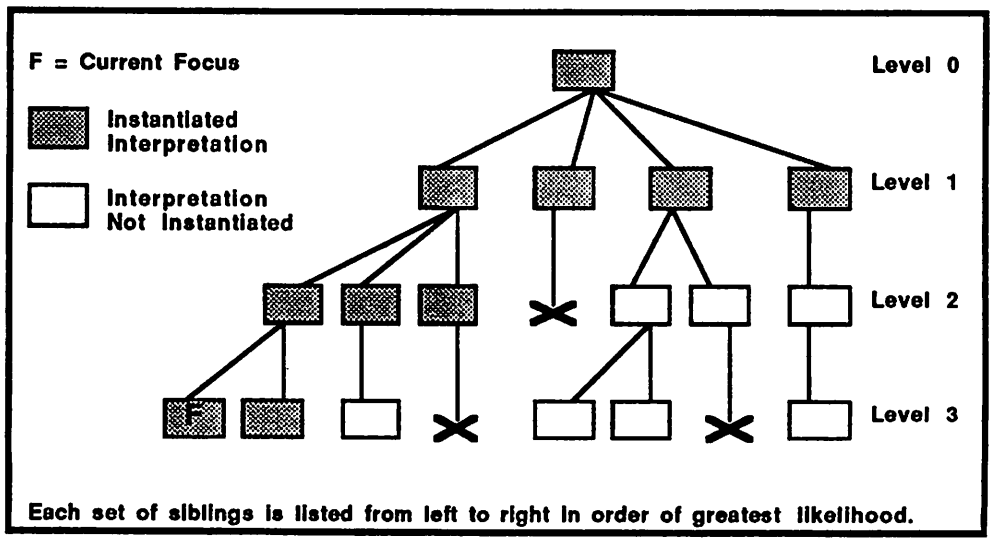

Figure 2.14 Focused Search

work that has been avoided completely (the non-shaded part of the tree has been drawn to show what would have been discovered if that work had actually been undertaken.).

It may happen that one of the focusing decisions later proves to be wrong, because heuristics are by definition fallible. Then either single level backtracking (Figure 2.15) or multilevel backtracking will occur (Figure 2.16). The better the focusing algorithms, the less backtracking and the fewer instantiated-but-discarded interpretations (shown in black).

In the figures (and in the examples in Appendix C), backtracking is chronological—the most recent focusing decision is selected for reconsideration. Whenever the current focus is found to be wrong, the focus is shifted to the most probable of its siblings; if there are no remaining siblings, the siblings of the parent are considered. Dependency-directed backtracking, an improvement over chronological backtracking, is possible if the implementation strategy of [Carver et al., 1984] is used.

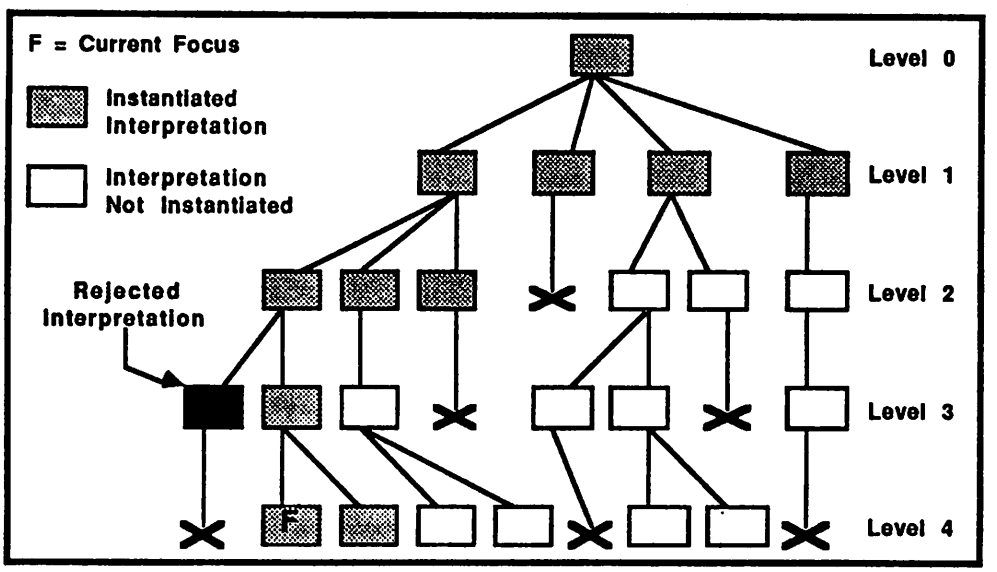

Figure 2.15 Single Level Backtracking

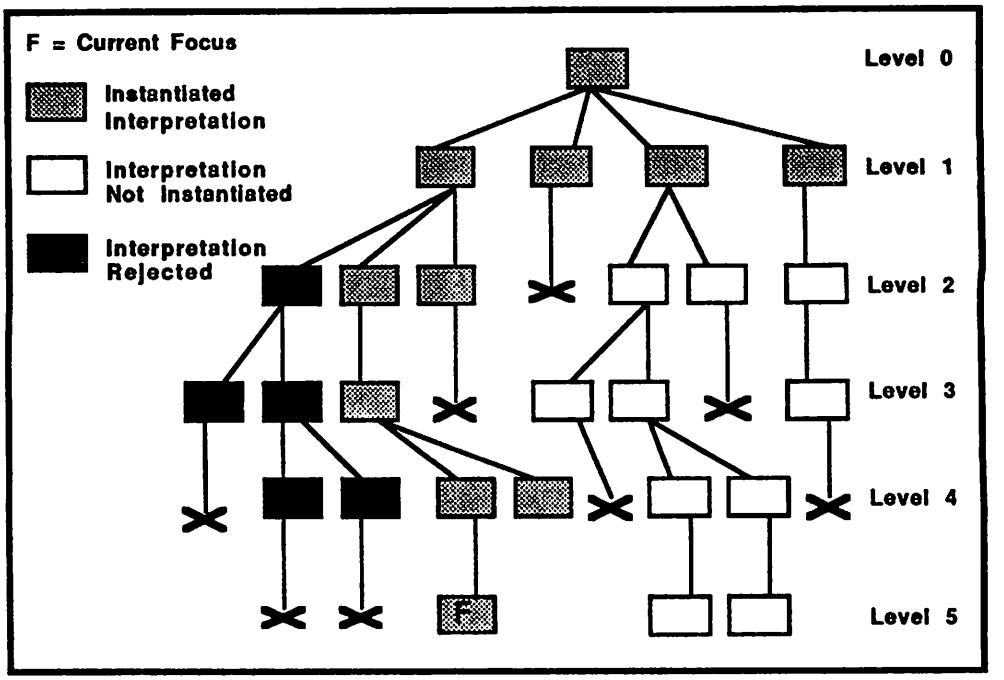

Figure 2.16 Multilevel Backtracking

2.2.2.4 Updating Status

After the viable interpretations have been instantiated, there is still work to be done to complete the interpretation process for each competing alternative. This work involves updating the state of the world and the state of the plan network to reflect the outcome of this action, so as to have a complete picture of the interpretation that is regarded as the "best" context in which to recognize the next action. By making an early focusing decision, we put off doing this work for all but the selected focus; if focusing is "perfect", then this work will in fact never actually be done except for a single interpretation at each level in the interpretation tree.

2.2.2.4.1 Updating Plan Network and World State

The updating process begins by asserting the effects of the action (primitive operator) in the world state. Then the path from the action to the top level operator is traversed to find any additional operators that may have completed. For each such operator, its effects are asserted in the world state, and its status is changed to "completed". At the same time, for each step on the path where operator O1 achieves condition C of operator O2, the condition is marked as satisfied; if it is the last pending precondition, then the subgoals of O2 are instantiated and the status of O2 is changed to "preconditions-satisfied".

In addition to processing the intended effects of this action on the world state and the state of the plan network, it is necessary to consider whether there were any "side-effects" of this action. This means visiting each pending precondition and subgoal in the plan network to see if any of these are now satisfied and should be marked as such. The preconditions are considered first. If there are any cases where a newly satisfied precondition is the last dynamic precondition that is pending, the operator status is updated to "preconditions-satisfied" and all its subgoals are instantiated. Then the pending subgoals are considered (including any newly instantiated ones). If there are any cases where a newly satisfied subgoal is the last pending subgoal, then the effects of the operator are posted and it is marked as completed. If the processing of the pending conditions resulted in any effects being posted, then the processing must be repeated in case there are any "side-effects" of the just completed operator's effects.

2.2.2.4.2 Protection Intervals

Actions, and their interpretations, are subject to one further validity test. The interpretation of an action should not violate any of the protection intervals in the plan network. Recall that a precondition is protected between the time it is achieved and the time the corresponding operator starts; and, a subgoal is protected between the time it is

achieved and the time the corresponding operator ends. The protection interval of a condition is violated when the effects of an action (either the immediate or the propagated effects) render the condition false.

To implement protection intervals, a list of "locked" conditions is maintained. Whenever a precondition or subgoal is marked satisfied, it is placed on the locked conditions list. Whenever an operator starts, its preconditions are removed from the locked conditions list; and whenever an operator ends, its subgoals are removed. After the status updating described in section 2.2.2.4.1 has been performed, the locked conditions list is checked to see if all these conditions are still true.

Violation of protection intervals should be avoided, as it leads to inefficient plans. It is not necessarily an error to violate a protection interval; when the violation is intentional, the condition must be taken off the protected list and marked as not-satisfied. But a violation could equally be an oversight on the user's part, or it could indicate that the focus is wrong. Thus, when a protection violation is detected, the user is given three options: to retract the action (violation was not intentional), to accept the violation, or to indicate that the condition should not have been on the protection list to begin with (here the focus must be wrong.) It is possible to extend the current algorithm so that an interpretation with a protection violation is "down-graded" in likelihood and its competitors without protection violations are preferred. This becomes another focusing heuristic, that cannot be applied until after the plan network is updated and the protection violations identified.

2.2.2.5 Least Commitment and Planning

The status updating algorithm of section 2.2.2.4.1 takes a least commitment approach to updating the plan network. The least commitment approach affects four situations. (1) If a condition is being tested to see if it is satisfied, then it is only considered to be satisfied if it evaluates true *without making any new parameter bindings.* (2) An iterated subgoal

that is iterated for a minimal count is *not* assumed to have completed once this minimal count has been achieved. (3) Similarly, no offline plans are assumed to have executed. (4) Finally, new parts of the plan network are not expanded until they are needed to explain a specific user action. Since the goal of plan recognition is to recognize the user's actual plan (which may be just one of several alternative plans), certain choices can be delayed until resolved by later user actions. The alternative to least commitment is to represent the possible alternatives explicitly; this could exacerbate the effort expended in search.

Take a situation where a new system S2 has just been defined and five directories exist that are not committed to any use as reference or working directories, and consider the commitment of a working directory for S2. It is possible, but not certain, that the user intends to use one of these five as the working directory for S2. The least commitment approach is to wait and see what the user actually does. The aggressive approach involves noticing that the subgoal *workdir-ready* of *setup-env* has a single achiever, *setup-workdir*; and, that the precondition to *setup-workdir* has a single achiever, *commit-workdir*; and that *commit-workdir* could be regarded as having completed for any of the five directories. This leads to six distinct possibilities each differing in both their plan networks and their world states. Five of the cases show *commit-workdir* completed, and *has-workdir(S2,d)* true for one of the five directories; the sixth case shows *commit-workdir* with a pending subgoal, and *has-workdir(S2,d)* not true for any directory d. There is clearly an advantage to the least commitment approach with its single interpretation over introducing five additional interpretations into the search tree.

If the least commitment approach is taken, then some later action will decide the issue. If there is an explicit *mkdir*, then everything is OK. However, a problem arises if the user starts to remove extraneous files from D, one of the existing, but not yet committed, directories; the action will fail the precondition *workdir-committed* of *setup-workdir*. But

this precondition could be satisfied if the (only) operator to achieve it were to be instantiated; that operator, *commit-workdir*, has no precondition and its subgoal and constraints are satisfied by D; instead of facing five possibilities for the working directory, the choice has been narrowed to a single case. Once *commit-workdir* is instantiated, we see that it can immediately complete, asserting in the world state the fact *has-workdir(S2,D)*, which is needed to satisfy the required precondition *workdir-committed*.

The correct implementation of a least commitment approach must involve more than "wait and see"; as the example above illustrates, simply delaying decisions is not enough. Thus, a failed dynamic precondition should not be immediately regarded as a reason to prune the interpretation. Rather, effort should be expended to see if the precondition is implicitly satisfied; only if this extra effort fails should the interpretation be pruned. Also, a static precondition that is not satisfied but which happens to match the goal of a plan in progress should trigger extra effort to reconsider whether or not that plan is actually done.

Satisfying a precondition to compensate for least commitment requires all the mechanism of planning—a plan is needed to achieve the precondition. However, we are not interested in executing actions to achieve the goal; we are interested in seeing if the goal is "already" satisfied. This could occur because there is an operator, not yet instantiated, whose subgoals are already satisfied; or, because there is a pending condition that is true for certain parameter values that have not yet been bound; or, it could occur by inferring completion of an iterated subgoal or execution of an offline operator; or, a combination of these factors could be involved. Therefore the planning that takes place must only involve complex or offline operators. Any planning algorithm is appropriate, provided this restriction is added. For the examples of Appendix C a very simple, linear planner was sufficient.

When there are competing interpretations, and some require planning while others do not, it is possible to use this distinction as an additional focusing heuristic. That is, those interpretations that do not require planning should be preferred, on the basis that they are the more "obvious" interpretations. In this way, the extra work of doing the planning will be avoided, as long as the heuristic is a good predictor. If the planning is eventually undertaken (either all alternatives require it, or the ones that didn't have now been ruled out for other reasons), the planner may fail to find a plan, in which case the interpretation is rejected outright.

2.2.3 Recognition Examples

Appendix C gives three plan recognition examples showing this recognition algorithm in action. These examples were run on the GRAPPLE system. Each example consists of an initial state and a sequence of actions, based on the operator library of Appendix A. The first example covers a basic case without complications. The second example shows how a protection violation is handled. The third example requires planning during plan recognition.

2.2.4 Algorithm Extensions

In this concluding section, we discuss two issues that are not handled by the algorithm presented here and one additional design issue. The first involves circular paths, and the second involves operators that achieve multiple goals. The additional design issue relates to search strategy.

2.2.4.1 Circular Paths

A circular path arises when it is possible for an operator to be used, directly or indirectly, to achieve one of its own preconditions or subgoals. A path is said to be circular when there are two or more instances of the same operator on the path. We are not aware

of any *reasonable* examples of circular paths in the software process. The *construct* operator in Appendix A has as its precondition that the environment be setup; *setup-refdir* is one operator that will be used in setting up the environment. *Setup-refdir* has a static precondition that the baseline be constructed. If this precondition were not static, then there would be a circularity involving the *construct* operator. The precondition has been designated static—not to avoid the circular path (although this is a by-product)—but to ensure that *construct* is always treated as a top level operator. That is, construction of a system version is considered to be a primary objective, which sometimes has the secondary effect of providing a baseline from which a future system version may be constructed. Thus, the circularity is avoided because of considerations of how the plans ought to be structured to capture a programmer's view of activities.

An example of a circularity from the blocks world is that unstacking a block, to make its top clear, can be used to satisfy its own precondition (with different bindings of course.) That is, if block X is stacked on block Y and block Y is stacked on block Z, then the goal of *cleartop(Z)* is achieved by *unstack(Y)* which has a precondition *cleartop(Y)* which is achieved by *unstack(X)*. This is a path involving two instances of *unstack* (one of X and one of Y).

The present algorithm does not handle such circular paths; these are detected at the time the achievers are calculated and an error message is issued. Because it generates paths top-down and does not use parameter bindings when it does so, it has no way to tell how many times to traverse the circularity before continuing with the rest of the path.

Inability to handle circular paths is not a particular liability. First, as we have mentioned, we know of no examples where this is needed for the software process domain. Second, using the well-known computer science technique of substituting iteration for (tail) recursion, it is possible to avoid circularities by definition of additional

complex operators with iterated subgoals. (This solution does however shift the burden to the operator writer.) In the blocks world example, one can define a new (complex) operator *unstack-many* with a single subgoal that is iterated for each block over x until *cleartop(x)*. Then instead of a path containing multiple instances of *unstack*, there are multiple paths each containing one *unstack* leading to a subgoal of *unstack-many*. This example is given in detail in [Huff & Lesser, 1987].

2.2.4.2 Operators Achieving Multiple Goals

The second limitation of the present algorithm lies in the way that it handles operators that achieve multiple goals. Each operator instance in a plan has a single achiever link to a precondition or subgoal one level up in the plan. This link captures the "purpose" of the operator. But, sometimes operators have multiple purposes. For example, this arises in the second example in Appendix C, where moving a particular file from one directory to another simultaneously serves to fill the reference directory and empty the working directory. This causes the algorithm to generate two interpretations, each of which shows a different purpose for the action, when in fact both purposes are simultaneously valid. Note however, that in each interpretation, both of the subgoals that the action achieves are actually marked as completed.

The problem with the present algorithm is not that it doesn't recognize both purposes being fulfilled; it does this correctly, even when the goals achieved are split across plans. Rather, the problem is that it generates two interpretations that are identical in all respects but one achiever link in the plan network. This is not actually a problem unless one of the interpretations eventually gets rejected; then any work expended on the second interpretation is guaranteed to be wasted, as it will eventually be rejected too. Unfortunately, to detect that the interpretations are duplicates requires additional work, and

so there has to be a trade-off decision as to when/whether this extra work should be undertaken.

2.2.4.3 Search Strategies

The algorithm presented here embodies one search strategy which is not the only option available. The most difficult issues in search involve the amount of search that should be done before reporting a possible error to the user, and the order in which alternatives should be searched. In the next paragraphs we discuss some variations on the current algorithm geared to these concerns, while in section 6.3.3 we discuss the need for further research in this area.

The user-interaction strategy described here is very simple. Basically, the recognizer assumes it always has the right interpretation, and thus reports problems to the user immediately. When the next action produces no interpretations from the current focus, the user is notified and queried to see if the action was really intended; the same thing occurs when there is a protection violation. However, in both cases, it is possible to search further for an interpretation in which there are no problems associated with the current action. Only if no such interpretation can be found, or a "reasonable" level of effort expended without results, should the user be notified of problems. The real issue is how much search to do before communicating with the user.

The heuristics that we have used to order the alternatives in search implicitly serve to compare interpretations at the same level in the interpretation tree. When only one alternative is picked at a given level, the heuristics will always be selecting among alternatives that have the same parent interpretation. In the absence of any other guidance, the focusing algorithm will exhaust all alternatives with the same parent before turning its attention to siblings of the parent, as was shown in Figure 2.15 and 2.16. This may not lead to the best search order, and two other approaches suggest themselves.

The first approach is to modify the current algorithm so that more than one alternative interpretation is carried forward from a given level; for example, when the heuristics are insufficient to indicate a unique choice, all the preferred alternatives could be used to generate interpretations for the next action. This entails more work so that more alternatives are available at a given level for the heuristics to choose among. Efficiency could be gained through the use of an ATMS [deKleer, 1986] to represent all preferred alternatives (and thus to share work common to one or more of them).

Another approach is to continue to follow only one preferred alternative, but to use additional heuristics that specifically address decisions across levels in the interpretation tree. We describe an example.

The example is diagrammed in Figure 2.17. Suppose that there is one interpretation I1 at level N, with two child interpretations (I2 and I3) at level N+1. If both I2 and I3 extend an existing plan and neither require planning (i.e., if the heuristics do not indicate a unique choice), then a random selection will be made between them. Suppose I2 is selected. Now, let there be two child interpretations (I4 and I5) at level N+2 with the parent I2. Suppose I4 extends a current plan while I5 starts a new plan, so I4 will be the initial focus for action N+2. If I4 has no successors at level N+3 (i.e., if I4 cannot be extended to accommodate action N+3), then there is an opportunity to make a cross-level comparison between I3 and I5.

Now it is possible to make a case for focusing on I3 instead of I5—I5 is less attractive because it involves starting a new plan. This is justified even though I5 already accounts for action N+2 while extra work must be done on I3 to recognize action N+2 (and there is always the possibility that I3 cannot be extended to account for action N+2). Perhaps the decision should be affected by the number of levels in which the competing alternatives differ. Here, I5 and I3 differ by just one level; a different decision could be made if they

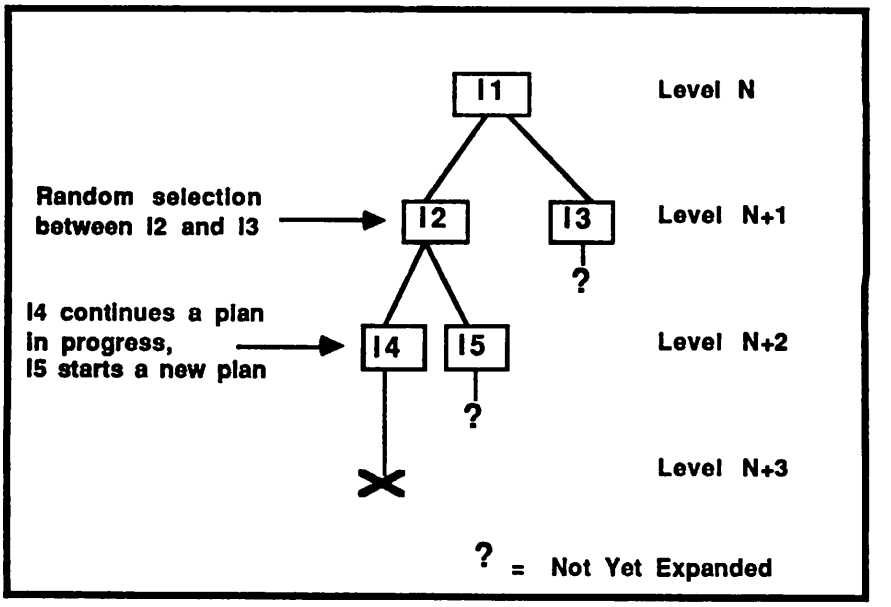

Figure 2.17 Cross-level Focusing Decisions

differed by two or more levels, on the basis that I5 can be explored more cheaply even though it is less attractive.

The issue is not implementing such a search strategy, for an implementation approach is given in [Carver et al., 1984]. Rather, the issue is finding a good set of heuristics. This is probably best done by running large numbers of recognition examples, across several different realistic operator libraries with real action sequences.

2.2.5 Computational Complexity

Planning and plan recognition are both inherently computationally expensive—several of the underlying algorithms are exponential. But the theoretical complexity of an algorithm does not necessarily give an adequate view of actual performance. For example, a search problem may involve exponentially many candidates to be searched; if there are ways to order the search using heuristics so that the most likely candidates are considered

first, then an otherwise intractable search becomes feasible. (Here performance is tied to the accuracy of the heuristics. If the heuristics are good predictors, performance will be very good; however, worst case behavior of the search will still be exponential.) As long as the heuristics are used to *order* alternatives, the algorithm is complete—if there is a solution it will find one. On the other hand, heuristics and restrictions may be used to *rule out* alternatives; then the algorithm is incomplete—it may fail to find a solution even though one exists. Incomplete algorithms can still be eminently useful; often, an otherwise unsolvable problem can be made solvable if the problem is slightly rephrased.

In the analysis that follows, we consider both theoretical complexity and the impact of heuristics and restrictions. There are three distinct parts to the plan recognition algorithm we have presented: the computation of the achievers relationships, plan recognition itself, and planning. Each is considered in turn.

2.2.5.1 Achievers Computation

The computation of the achievers relationships takes place in advance, before any actions are seen, and thus does not affect the "real-time" performance of the recognizer. It involves viewing expressions in the operator definitions as expressions in first order logic with implicit quantifiers. As with most interesting algorithms for first order logic, resolution, which is the theoretical basis of the achievers computation, is NP-complete, for which the only known algorithms are exponential. (Actually, the theoretical situation is somewhat worse—resolution is not guaranteed to halt; if halted arbitrarily, exponentially much work will have been done.)

In GRAPPLE, which is implemented in Prolog, the achievers computation is accomplished by using the logical mechanisms (unification and resolution) that are built into Prolog. Each operator is considered in turn. Its goal is asserted into the Prolog database, using a unique constant for each variable in the goal. Then an attempt is made to

"prove" each subgoal and dynamic precondition; this is done by posing the subgoal or dynamic precondition as a Prolog goal. A successful proof is indicated when Prolog returns the answer "yes" (along with an assignment of variables in the subgoal or dynamic precondition to the unique constants). Each successful proof thus identifies both a new achiever relationship (between the operator and the subgoal or dynamic precondition that was proved) and the corresponding parameter map.

In the absence of computed predicates, the proof is accomplished directly by unification. In the presence of computed predicates, resolution may be involved as well. The logical content of the definitions of the computed predicates is "grist" for the resolution "mill". For example, suppose operator A has a goal of f(x,y) and operator B has a subgoal g(a,b) and there is a computed predicate which defines f(x,y) as "g(x,y) and h(x,y)". In order to detect that operator A can achieve the subgoal of operator B, the fact that g(x,y) logically follows from f(x,y) must be available. This can be expressed directly in Prolog as the rule "g(x,y):-f(x,y)". In the present implementation, these rules must be explicitly included with the operator library—they are not generated automatically from the computed predicate definitions. In the operator library of Appendix A, no rules were needed even though there are computed predicates.

The number of (top level) Prolog goals to be proved is equal to the number of operators in the library times the number of dynamic states in the library (recall that each subgoal or dynamic precondition is a dynamic state). The work required for each proof is trivial if only unification is involved; when there are rules to resolve with, the work is related to the complexity of the rules involved. (Note that, for pragmatic and performance reasons, the inference mechanisms of Prolog are both incomplete, due to the use of depth-first search in choosing the clause by which to do the reduction, and unsound, due to the omission of the occurs-check in unification. This places the burden on the designer of the operator library to follow normal Prolog rules in expressing the deductive rules that will be

used in resolution; for example, rules for base cases should be listed before the rule with the recursive step.)

Computing the achievers for the operator library of Appendix A, which has 15 operators and 16 dynamic states and uses no deductive rules, takes place essentially instantaneously in GRAPPLE.

2.2.5.2 Plan Recognition

Plan recognition is fundamentally a heuristic search through a tree of interpretations (such as that in Figure 2.14). The heuristics are used to order the search, so the worst case performance is a function of the number of interpretations to be searched, which is potentially exponential, as discussed below. Expected performance is difficult to quantify because the heuristics and their predictive quality are also domain-dependent; performance will be better in some domains because of the availability of better heuristics.

2.2.5.2.1 Search Space

The size of the search tree (such as the one diagrammed in Figure 2.14) determines worst case performance (which has more theoretical than practical importance). A look at Figure 2.14 shows that the number of interpretations to search is potentially exponential in the number of actions. In fact, Figure 2.14 represents a tree that has already been pruned—interpretations that fail the viability tests (see section 2.2.2.2) are not included in the tree. (These are interpretations which can be shown to be incorrect based on the information available from the actions seen to date; interpretations that pass the viability tests may be incorrect, but cannot be ruled out until more information, in the form of more actions, is available.)

The actual size of the tree, including both interpretations that pass and that fail the viability tests, is related to the characteristics of a given operator library and the number of

plans that are simultaneously in progress. Tree size is a function of the branching factor, the number of children of a given parent in the tree. A key component of the branching factor is the number P of possible paths from a given action (i.e., primitive operator) to a top level goal. In the operator library of Appendix A, the maximum value of P occurs for the operator *move* and is four; recall that these paths can be read off Figure 2.6 by following links from bottom to top. The distribution of values of P, and the maximal value of P, for a given operator library gives one view of complexity for plan recognition purposes; the lower the value of P for a given primitive operator, the less uncertainty there is in recognizing that operator.

If the actions being recognized were such that only one plan was known to be in progress at a time, then the maximal value of P would be the maximal branching factor. However, if there are M plans in progress, then the branching factor is a function of P*(M+1). That is, each path must be considered as a possible continuation of each existing plan as well as the start of a new plan. (The rather significant difference between a branching factor of P and one of P*(M+1) indicates the additional complexity that arises from making the seemingly simple assumption that multiple plans can be in progress concurrently.) The actual number of possible interpretations identified prior to making the viability tests will depend on the characteristics of the current plans. Some of the P*M possibilities for continuing plans will not be syntactically legal; the path must connect to an instantiated but not yet satisfied subgoal or dynamic precondition in order to legally continue a plan in progress.

The actual growth in the tree depends not just on the branching factor, which characterizes how many new nodes are created, but on a decay factor, which characterizes how many of those nodes need not be carried forward. The decay factor is a function of the ability to bind parameters early and thus to rule out interpretations with the viability

tests. Unfortunately, the decay factor will be dependent on the set of operators, the actions being recognized, and the successive states of the world during recognition.

One can imagine a worst case plan recognition problem where the decay factor is negligible so that the tree in fact grows exponentially. Thus, we have to say that the search in plan recognition, in the absence of heuristics, is potentially exponential in the number of actions. On such a worst case problem, one or more plan parameters that are necessary to disambiguate interpretations cannot be bound until the last action in the last concurrent plan occurs. Only when this last action is seen will it be possible to prune the tree down to a few competing alternatives (or to a single alternative) that explain the sequence of actions seen.

However, many plan recognition problems will not be as "ill-conditioned" as this. In a "well-conditioned" plan recognition problem, the observation of a few actions will be all that is needed to narrow the search to a few possibilities. While these early actions are taking place, the tree will first grow exponentially and then decay to a few alternatives; thereafter there will be no further growth, and perhaps some additional decay. In such a problem, the search space will be at worst exponential in a number Q, where Q is much smaller than the number of actions.

This analysis shows that plan recognition (in the absence of heuristics) involves exponential search in the worst case. The actual amount of work required for search depends on how well- or ill-conditioned a given problem is. This cannot readily be measured for a given operator library in advance, but can be observed by running some number of example plan recognition problems.

In the operator library of Appendix A, there are seven primitive operators; three (*define_system*, *construct*, and *create*) have single paths, three (*mkdir*, *copy* and *delete*) have two paths, and one (*move*) has four paths. If the user is in fact only executing one

top level plan at a time, then the maximal branching factor is 4*(1+1) or 8. On a small number of test examples run, at most two (of at most eight) possibilities survived the viability tests, giving an effective branching factor of two. All these problems were well-conditioned, and typically no more than three or four actions were required to *narrow the* search to a single interpretation. Three of these examples are given in Appendix C, and the search tree for one of them appears in Figure C.1.

2.2.5.2.2 Search with Heuristics

The preceding analysis addresses the problem of search without heuristic guidance, and shows clearly why heuristics are necessary to practical plan recognition.

The most interesting measure of plan recognition performance is, somewhat unfortunately, one that can only be measured empirically (on a given set of problems for a given operator library). This is a measure of the effectiveness of the heuristics, which can be computed as the number of interpretations that were ever chosen as the current focus divided by the number of actions recognized. If the heuristics are "perfect", minimal work will have been performed and this measure will be 1; if the heuristics lead to wrong choices that must later be reconsidered, then the measure will be greater than 1. For the search tree given in Figure C.1, the heuristic effectiveness is 1.17—there was one wrong focusing choice over the course of six actions. As we have already said, this measure characterizes performance for a specific set of heuristics in a specific domain.

If the heuristics available for a given domain do not prove to be very effective for focusing on the right interpretations, there are other methods to explore. Specifically, it is possible to solicit information from the user to help the focusing. Obviously, this makes the intelligent assistant more obtrusive, and care must be taken that it not become "too" obtrusive. The intelligent assistant must therefore make considered decisions about *when*

to get additional discriminating information from the user and *what* information should be requested. These are avenues for further exploration.

2.2.5.3 Planning

The complexity of planning is of interest here for two reasons: first, because we have used a planner as a subsidiary component in plan recognition, and second, because it is a component of the intelligent assistance architecture in its own right (independent of any role it may play it plan recognition.) Note that when planning is called by plan recognition, the amount of planning that needs to be done will usually be small: the plan may already exist partially or in its entirety, significant parts of the plan have already been achieved, and only complex and offline operators need to be considered in any further development of the plan.

Planning presents multiple issues of algorithmic complexity. Nonlinear planning was originally proposed to overcome the problem of exponential search among possible orderings of actions. But, the truth-criterion which is central to nonlinear planning has been shown [Chapman, 1987] to be NP-complete (i.e., effectively exponential) for all but severely restricted forms of operator definitions. Several other sub-issues that arise in planning are combinatorial [Wilkins, 1988], such as ordering actions to handle conflicts, constraint satisfaction, and resource allocation.

A planner which has dealt with these issues in a practical way is SIPE [Wilkins, 1988]. The identification of trade-offs, and the development of heuristics and restrictions, to achieve efficiency while still solving a useful range of problems has been a major focus of the SIPE work. SIPE is neither sound nor complete: that is, it can produce incorrect plans (although this can be detected at some additional cost) and it may fail to produce a plan even though one does exist. On standard problems, it is more efficient than other planning systems; on complex problems, it provides solutions where other systems cannot.

Data from SIPE can be used to calibrate the practicality of current planning technology. In a mobile robot domain with 25 operators, 25 "deductive operators", and 200 predicates, plans that involve 170 goal nodes can be produced in 30 seconds on a Symbolics 3600. (It only takes about 9 seconds for the planner to produce a partial plan that the robot can start executing.) Domains in which the plans are far simpler (30-50 goal nodes), even though the number of operators and predicates may be equal or greater, can be handled in 5-8 seconds.

2.3 Summary

In this chapter we have presented a basic language for operator definitions and a basic algorithm for plan recognition. The setup-environment task was used to highlight specific language issues that arise in operator definitions for a typical set of software activities and to showcase one set of language features designed to handle these problems. The requirements of a plan recognition algorithm for the intelligent assistance application were discussed and a suitable algorithm described. A novel feature of this algorithm is the use of a planner in support of plan recognition functions.

The purpose of this chapter has been to establish a clearly defined foundation on which the research to be described in the following chapters is built.

CHAPTER 3
DEEPER DOMAIN MODELING

In this chapter, we show that intelligent assistance for software development involves *hidden state*, where useful information about the state of the world is missing. This proves to be a significant barrier to achieving a nontrivial representation of the domain and leaves the intelligent assistant without a basis for independently critiquing many actions of the user. We advance a solution to acquiring additional state information by using domain knowledge to make plausible assumptions about the missing values based on the observed state. This process is formalized as nonmonotonic reasoning. Using these plausible assumptions, the *credibility* of competing alternatives can be evaluated independently of the user; actions that are consistent with current assumptions will have the highest credibility. If it becomes necessary, actions that have low credibility can still be pursued after *reconciling* the assumptions with the requirements of the actions. The ability to make plausible assumptions allows additional domain knowledge about the context for actions to be made explicit; thus the approach enables deeper modeling of the domain.

In the first section we define the problem and give an overview of the entire approach. In section two, we given additional examples of applying this technique to software development processes. In section three we discuss in depth the technical issues that affect

how domain knowledge is formulated with nonmonotonic rules. The purpose of this chapter is to define the problem, describe the approach, and discuss technical issues that arise in applying this approach. The detailed aspects of implementing the approach are presented in Chapter 4.

3.1 Approach to Deeper Domain Modeling

Substantive support of processes requires involvement in both the complex decisions as well as the mundane details. We will show that formally representing the knowledge involved in some of these decisions can be a challenge. As examples in the software development application, consider the criteria for choosing the baseline from which to develop a new system version, selecting tests to run, or deciding which system version is releasable. If a process assistant lacks knowledge to address these decisions, it cannot independently critique a choice made by the user, nor can it suggest a restricted set of likely candidates from which the user can choose. In this case, intelligent assistance is seriously restricted because the representation of the domain is limited to surface issues.

3.1.1 The Hidden State Problem

Consider the goal of testing a new system version. To test a system means running all applicable testcases, and only those cases (neither under- nor over-testing is desirable). The problem is that applicability of testcases cannot be directly observed as a result of past actions nor computed with certainty from observable data. Thus, when a testcase is run, there is no independent basis for determining if the testcase is indeed applicable, or if this is the last of the applicable test cases (which signals the end of testing). Knowledge about testing strategies (where a given strategy implies that certain categories of testcases are applicable) underlies the choice of testcases, but this deeper knowledge cannot be exploited since the operative testing strategy cannot be determined—it too is not observable.

When information about the state of the world is incomplete, we say that a planning application involves *hidden state*. The software development application involves hidden state since predicates indicating whether a testcase is applicable or whether a system version is releasable are not directly observable, nor are they readily computable from the observable data. Hidden state presents significant problems. When it is impossible (or infeasible) to acquire the missing information, plan recognition is adversely affected. Competing interpretations of actions cannot be disambiguated, some distinctions between legal and illegal actions/plans cannot be made, predictions of future actions lack precision, and there is no basis to explain why observed actions/plans were chosen over other alternatives.

These problems with planning in the absence of complete state information are due to the difficulty of achieving accurate descriptions of operators. Predicates whose truth or falsity cannot be determined cannot be used in operator definitions to define the relevancy of an operator (preconditions), the decomposition and completion criteria (subgoals), or restrictions on parameter bindings (constraints). When such predicates are omitted entirely, the operator definitions are under-constrained and plan recognition is too permissive—"illegal" plans will be accepted, predictions will be too general, and alternatives that are actually irrelevant cannot be discarded. Attempting to compensate by substituting another expression for the missing predicate may yield a definition that is over-constrained and plan recognition will be too rigid.

In intelligent assistance, as in other applications of plan recognition with hidden state, it is important to find some balance between these two extremes of permissiveness and rigidity (as illustrated in Figure 3.1). An approach is needed that will closely approximate the desired situation, without being consistently too permissive or consistently too rigid. Further, the approximation should be "elastic", in the sense that it can be adjusted when a deviation is identified. This is shown in Figure 3.2.

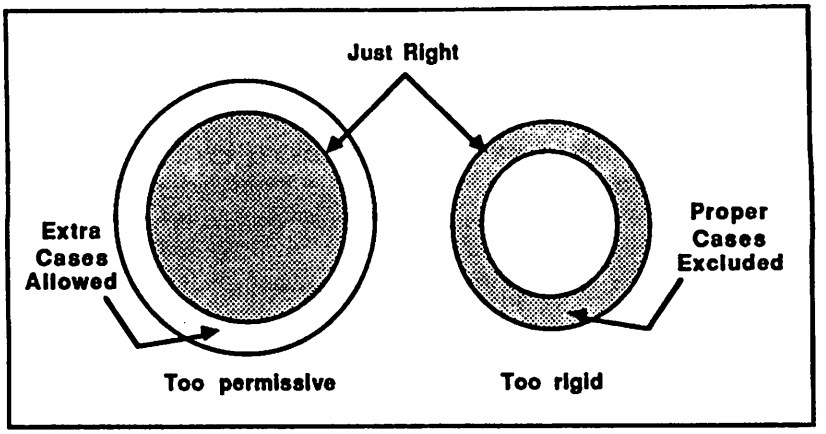

Figure 3.1
Rigidity versus Permissiveness in Plan Recognition

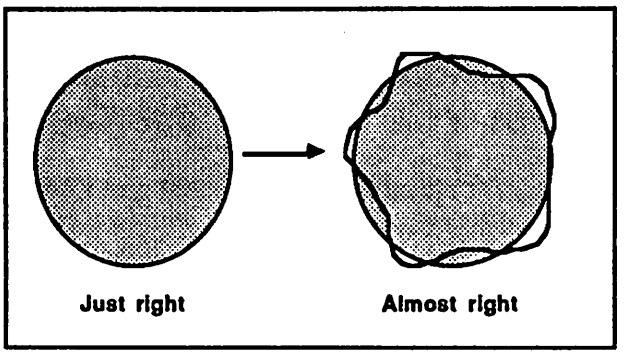

Figure 3.2
Finding a Balance between Rigidity and Permissiveness

3.1.2 Towards a Solution

One approach that captures the spirit of Figure 3.2 is to use the observable state information to make plausible assumptions about the missing state information. In the testing example, there is a correlation between the types of changes made during source editing (something that can be measured) and the operative testing strategy, which in turn

determines the categories of tests that are applicable. For example, when changes are simple (affecting a few lines of code), a weak test strategy would typically be appropriate—only base testcases would be applicable. Otherwise, standard testing would typically be appropriate—base and normal testcases would be applicable. This reasoning is inherently nonmonotonic. Assumptions are made in the absence of information to the contrary; later, additional information may be acquired that defeats the earlier conclusion and its consequences.

When assumptions about test strategy and applicability are added to the observable state of the world, it is possible to evaluate the *credibility* of alternatives. Interpretations of actions (or predictions of future actions) that agree with the current assumptions have the highest credibility; interpretations lose credibility with each assumption that is violated. If the operative testing strategy is assumed to be standard testing, then an action to run one of the normal cases is fully credible, because that case is assumed to be applicable. On the other hand, an action to archive the system, which is predicated upon testing being completed, would have less than full credibility as long as there are applicable (i.e., base or normal) cases still to be run.

Given two alternative interpretations that differ in credibility, the more credible alternative is more likely to be the correct interpretation. Given two choices for completing an unsatisfied subgoal, the more credible alternative is the better prediction of the future. An action whose "best" interpretation is below a certain credibility threshold is a possible user error. Thus, the programmer can be advised of a possible oversight when archiving prior to running all applicable testcases. Finally, it is possible to give the underlying reason for the credibility of running or not running a particular testcase, by citing the operative testing strategy and its implications.

Credibility can be combined with other discriminators to determine which interpretations to pursue. Sometimes it will be necessary or desirable to pursue an interpretation that is not fully credible; for example, the interpretation may still be the "best" considering all available discrimination information. In order to proceed with such an interpretation, it is necessary to *reconcile* the assumptions about the state of the world with the requirements of the desired interpretation. For example, suppose the operative testing strategy is assumed to be standard testing. To pursue an interpretation in which archiving starts when only base cases have been run, it is necessary to revise the assumption that testing is not done. While this can trivially be accomplished by simply recording that testing is done, it is far more interesting to provide some rationale for testing being done. And, indeed, the preferred reconciliation is to change the operative testing strategy from standard to weak testing, after which it follows that testing is now done. (The full implementation of all the examples informally described here will be given later).

In the remainder of this section, we show how a plan recognition system, based on the classical hierarchical planning paradigm, can be extended to incorporate a new type of domain knowledge about the context for actions; unlike the knowledge already reflected in operator definitions, this knowledge is approximate rather than absolute. This approach allows plausible inference and plausible explanation within a deeper model of domain activities than is otherwise possible. In section 3.1.3, we show how to capture this deeper knowledge, and how to exploit it for reasoning about world state. This is accomplished through monotonic and nonmonotonic rules in a truth maintenance system (TMS). In section 3.1.4, we explore the impact on a plan recognition architecture. *Credibility* represents a new perspective from which competing interpretations can be disambiguated. *Reconciliation* is the means by which assumptions are revised when the plan recognizer discovers that its assumptions are wrong.

3.1.3 Reasoning about the State of the World

Filling in the missing details about the current state of the world is a process of extrapolation from what is known about the state. Extrapolation involves adding certain specific conclusions about the state when a particular pattern of other propositions holds in the state. Thus, extrapolation lends itself naturally to a rule-based process: *when <pattern>, add <conclusion>*. In this approach, the additional knowledge that enables deeper domain modeling is captured in these rules, not directly in the operators (which are the standard vehicles for expressing domain knowledge). Thus, actions determine a *core* state; the rules are then applied to the core state to arrive at an *extended* state. If the additional knowledge were to be expressed directly in the operators, individual rules would have to be replicated in multiple operators. (A pattern can be an arbitrarily complex logical expression involving predicates whose truth is determined by many different operators; each such operator would have to use the pattern in a conditional effect.)

It is a requirement that the rule system support nonmonotonic reasoning, in order to capture the conjectural nature of some of the conclusions. Nonmonotonic reasoning allows a connection between pattern p and conclusion c that is a special kind of logical implication; this connection between p and c is such that p *typically* implies c (that is, *if there is no information to the contrary*, then c holds if p holds.) Different systems for nonmonotonic reasoning, including circumscription, modal logics, and default logic, take different approaches to formalizing the concept of "no information to the contrary". For an overview of nonmonotonic reasoning, see [Genesereth & Nilsson, 1987].

A truth maintenance system (TMS) [Doyle, 1979] is one approach to implementing nonmonotonic reasoning, based on multi-valued logic. A TMS maintains a network of nodes, each of which can be labelled IN or OUT. Separate nodes are used for a proposition and its negation. (A proposition is a predicate with bound arguments; if the

predicate is a core state predicate, then the proposition is a core state proposition.) If the node for a proposition is IN and the node for its negation is OUT, the proposition is true; if the node is OUT and the negation is IN, the proposition is false. If both are OUT, the truth value is unknown; if both are IN, there is a contradiction.

Justifications capture the relationships between the nodes, correlating a set of support nodes and a set of exception nodes with a conclusion node. A justification of the form A EXCEPT B —> C means that if A is IN and B is OUT then C is IN. The exception node B represents the nonmonotonic content of the justification; a monotonic justification has an empty list of exceptions. In order for a node to be IN, it must have at least one valid justification; a justification is valid if all its support nodes are IN and all its exception nodes are OUT. A premise justification has empty support and exception lists, so it is always valid.

3.1.3.1 Domain Knowledge in TMS Justifications

As an example, take the selection of operators in Figure 3.3 for building a system version. The effects of these operators determine the core state of observable facts. The use of the extended state predicates that cannot be observed is highlighted. For example, the precondition in *run_one_case* requires that the testcase be *applicable*; the iteration completion criteria in *run_cases* is that *tests_done* be true; and, there is a subgoal in *build* that will allow archiving to be skipped when waiving it is appropriate. The objective is to use TMS justifications to derive truth values for these extended state predicates.

Example domain knowledge about *applicable* and *tests_done* is given in justification form in Figure 3.4. In order to express this knowledge, several additional predicates have been introduced. Changes made during editing are used to conjecture whether the test strategy should be standard or not (rules J1-J2). The nonmonotonic rule J1 can be read "In the absence of information to the contrary, if substantive changes are made during editing,

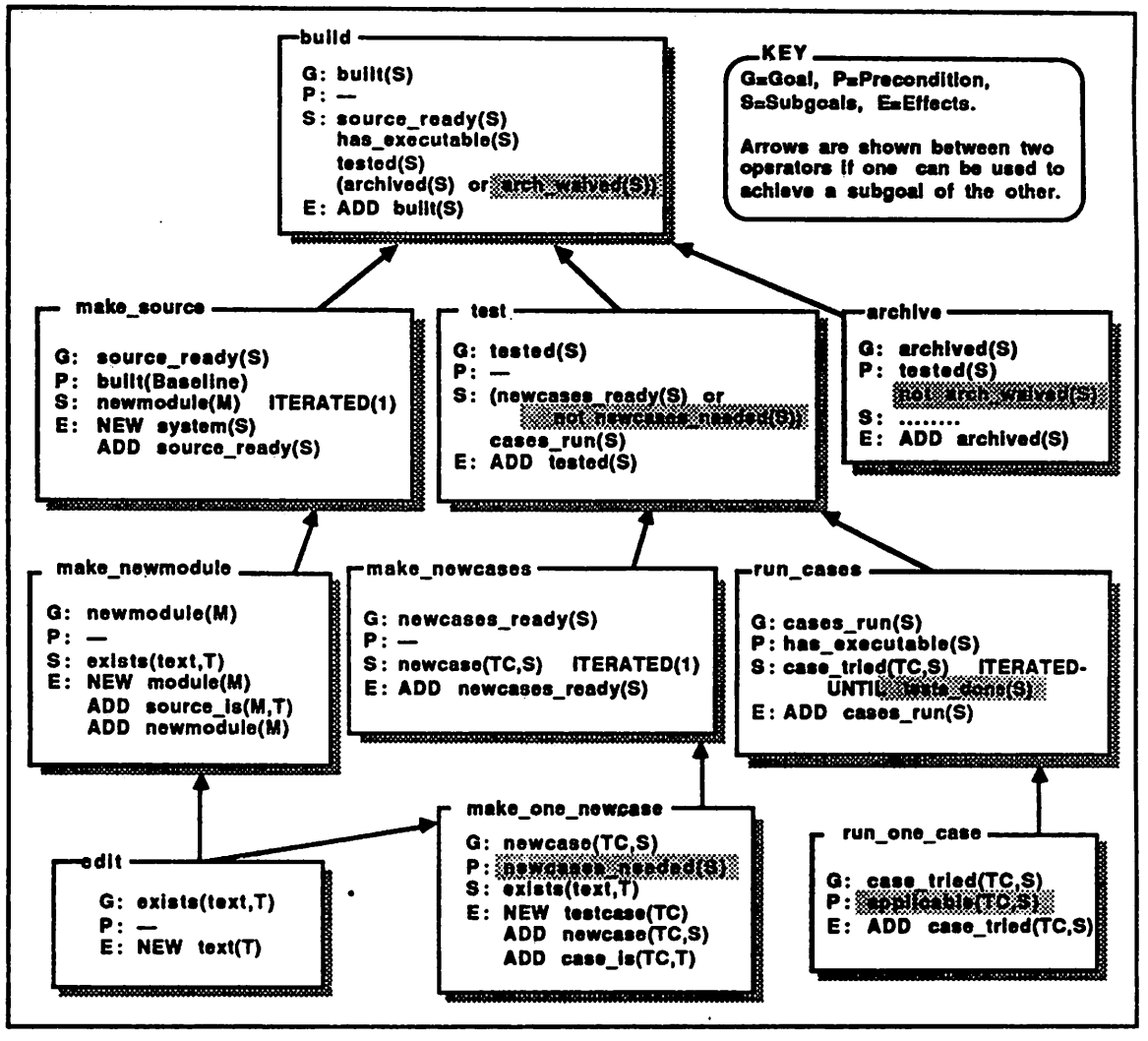

Figure 3.3 Operators for Building a System Version

then a standard testing strategy is appropriate." Testing strategy determines whether normal testcases are relevant (rules J4-J5); base cases are always relevant (rule J3). Rules J6-J7 establish that cases that are relevant are applicable unless they are specifically excluded (as they would be if infected by some catastrophic bug that is not yet fixed). Rule J8 states that typically cases are not specifically excluded. *Tests_done* does not hold until all applicable testcases have been tried (J9-J10). Note that both monotonic and

```
Operative Test Strategy
    J1: substantive_changes(Sys) EXCEPT not standard_test(Sys) —> standard_test(Sys)
    J2: EXCEPT standard_test(Sys) —> not standard_test(Sys)      %weak testing = not standard_test

Relevancy of Cases, By Category
    J3: type(Case,base) —> relevant(Case,Sys)
    J4: type(Case,normal) and not standard_test(Sys) —> not relevant(Case,Sys)
    J5: type(Case,normal) and standard_test(Sys) —> relevant(Case,Sys)

Applicability of Cases
    J6: relevant(Case,Sys) and not spec_excluded(Case,Sys) —> applicable(Case,Sys)
    J7: EXCEPT applicable(Case,Sys) —> not applicable(Case,Sys)

    J8: EXCEPT spec_excluded(Case,Sys) —> not spec_excluded(Case,Sys)

Completion of Testing
    J9: applicable(Case,Sys) and not case_tried(Case,Sys) —> not tests_done(Sys)
    J10: EXCEPT not tests_done(Sys) —> tests_done(Sys)
```

Figure 3.4 Example Justifications

nonmonotonic justifications have been used; the meaning of the testing strategies is defined monotonically, for example.

These justifications are predicated upon being able to determine the extent of changes made to the source code for a system during editing. The easiest way to do this is to measure the difference between the source of the baseline for the system and the source of the system, which can be done at the time of linking (when it is known for certain that all source editing is completed.) That means that the link action should actually be a command language script that both calls the link editor and performs the difference operation. In this way, *substantive-changes* is a predicate that can be set by the effects of the *link* operator .

(This example takes a rather simplistic view of determining test strategies and test case applicability. For a discussion of the relevant domain knowledge that would make this example more realistic, see section 3.2.4. Also, the process of translating domain knowledge into justifications is treated in detail in section 3.3.)

TMS justifications bear a strong resemblance to rules in default logic [Reiter, 1980], an alternative, and more general, approach to nonmonotonic reasoning. In default logic, a rule is written A:MB/C to mean that if A is provable and B is consistent (that is, ¬B is not provable), then C can be concluded. A TMS translation would be A EXCEPT ¬B —> C. This simple translation hides fundamental differences, and their practical implications. The TMS deals with validity (truth in a model), while default logic deals with provability (validity in all models); for the approach to hidden state described here, the TMS facilities are adequate. (The formal relationship between the two systems has yet to be studied in detail [Etherington, 1987].) As an example of practical implications of these differences, consider a rule set in default logic and the equivalent rule set expressed in TMS form. Since an extension is the default logic concept corresponding to a TMS labelling, we might expect the extension and labelling to give the same answers. However, they do not always do so: for example, there are rule sets having multiple extensions and exactly one labelling [Morris, 1987].

3.1.3.2 Representing World State in the TMS

Justifications provide a way to derive the extended state from the core state. When the justifications are instantiated, and the truth values for core state propositions entered (with premise justifications), the truth maintenance process will label the nodes, giving a read-out on the truth values for the extended state propositions. When the core state changes as a result of an action, the premises will change, nodes will be relabelled, and the extended state propositions may change; in this way, the extended state propositions may vary over time. The TMS will also determine whether the truth value of a given proposition is *certain* or *by-assumption*. A proposition is certain unless one or more nonmonotonic rules were used to determine its truth value. (The truth values of propositions that are certain cannot be changed in order to reconcile an interpretation.)

As an example, consider a situation where the building of system version SV has progressed to the point where testing is in progress; suppose also that the source editing changes were substantive. Let there be three testcases, where TC1 is a base case, and TC2 and TC3 are normal cases; suppose TC1 has been run. The state of instantiated justifications is given in Figure 3.5, showing that standard testing is assumed to be operative (conclusion of J1), that TC2 and TC3 are assumed to be applicable in testing SV

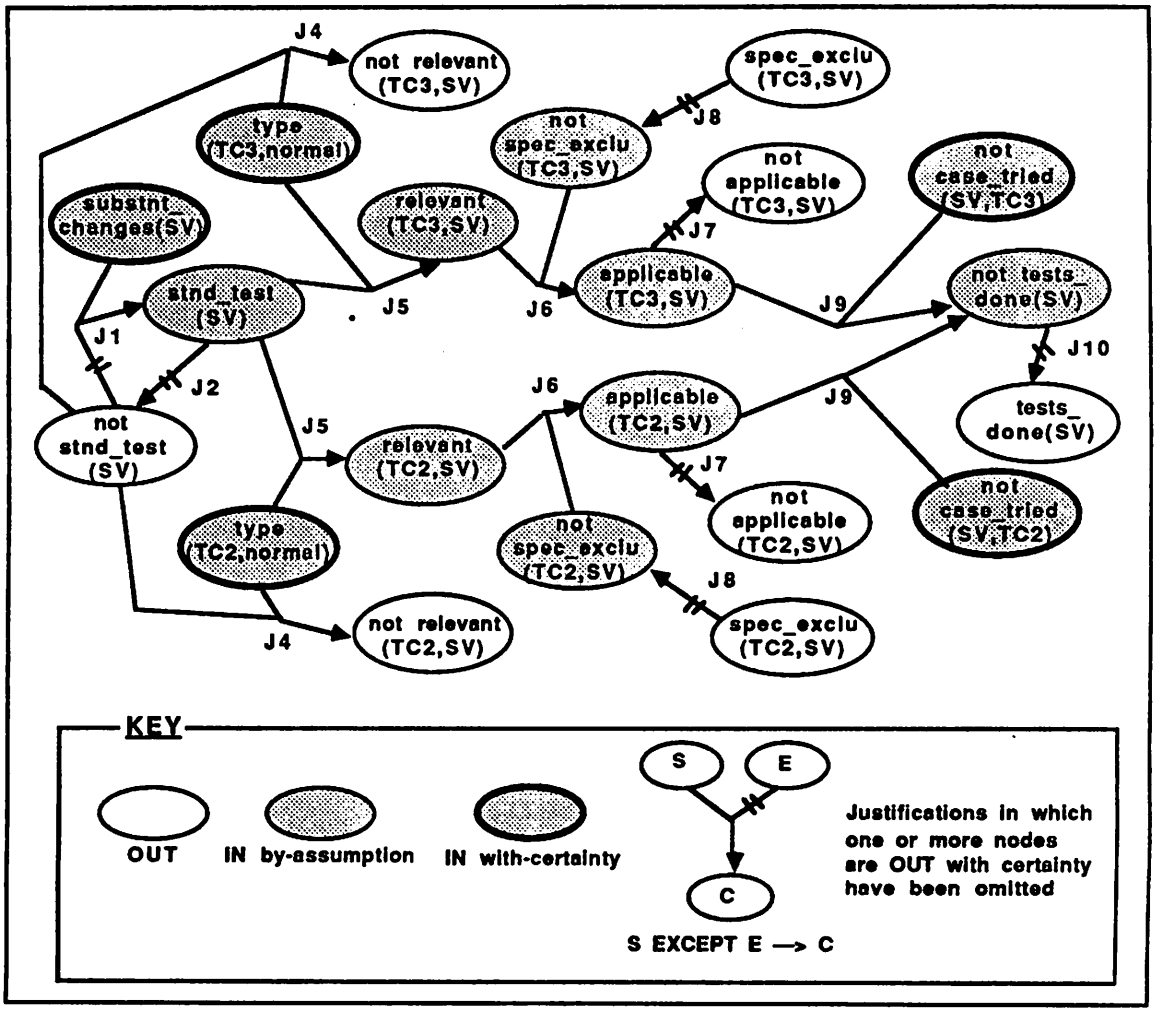

Figure 3.5 Instantiated Justifications

(conclusions of two instances of J6), and that testing is not done as there are (two) applicable cases that have not yet been run (conclusions of two instances of J9).

It should be noted that this use of a TMS is somewhat different from traditional uses. In a traditional TMS application, the problem solver using the TMS is constantly discovering new justifications; this discovery process is guided by the current state of the TMS. At any given time, the current justifications represent only partial knowledge concerning the interrelationships among the facts about the world. And, the TMS is used to model successive approximations of the same state of the world, not successive states. Our use of the TMS differs in two ways. First, all justifications are predefined; there is no dynamic discovery of new interrelationships expressed in new justifications. Second, the TMS is used to model successive states of the world (shown clearly by the fact that premises can be retracted.)

3.1.3.3 Implications for Deeper Domain Modeling

We have shown how justifications and the labelling mechanism of a TMS can be used to extrapolate from observable information to fill in missing facts about the state of the world that are needed to define domain operators. We need to characterize the nature of the domain knowledge that is captured in the justifications in order to understand why this approach allows deeper domain modeling. We do so by first discussing some of the characteristics of the example we have developed, and then generalizing.

The justifications shown in Figure 3.4 are all directed at supplying truth values for the predicates *applicable* and *testing_done*; these predicates are needed in the operators *run_one_case* and *run_cases* respectively (as shown in Figure 3.3). It turns out that neither of these predicates can be estimated directly from observable predicates. Rather, there is a whole chain of reasoning that leads to approximating the predicate *applicable*, and that predicate is then available to be used in the approximation of *testing_done*. This reasoning,

embodied in justifications J3-J7 and J9-10, captures background knowledge about the programming process domain. In this case, the background knowledge involves such notions as testing strategies, and how the operative testing strategy affects test case selection.

In order to be able to apply this background knowledge, other predicates have to be approximated; without a way to evaluate *standard_test* or *spec_excluded*, the background knowledge cannot be exploited. The justifications that do this (J1-2 and J8) capture empirical relationships. Justification J1, for example, cites a correlation between substantive editing changes and the standard testing strategy, without explaining the correlation in any way.

From this description, we can see that the knowledge captured in the justifications is of two sorts: background knowledge capturing fundamental principles about the domain, and empirical knowledge about observed relationships. As shown in Figure 3.6, the background knowledge by itself is incomplete, and the empirical knowledge is needed to

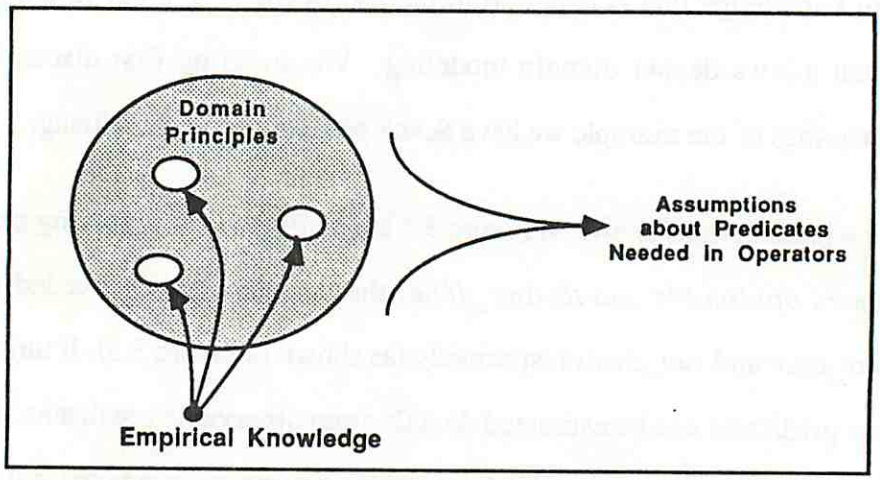

Figure 3.6 Knowledge in Justifications

fill the gaps. It is the background knowledge, not the empirical knowledge, that actually contributes to deeper domain modeling; if only empirical knowledge were involved, there would be no deeper understanding of the domain. The empirical knowledge is the source of the uncertainty in the reasoning. (In our example, the two justifications J7 and J10 are in nonmonotonic form for convenience; the uncertainty in *applicable* and *tests_done* actually arises from justifications J1, J2 and J8.)

Although the primary objective of capturing additional domain knowledge was to define selected predicates that were needed in operator definitions, the approach we have taken has actually resulted in the definition of a much larger set of new domain predicates, as well as the definition of relationships (theoretical and empirical) among these predicates. Many of the new predicates do not appear directly in operator definitions, but their truth status affects that of predicates in the operators. In this way, the justifications capture knowledge about the context for actions and allow plausible inference within this previously inaccessible context.

3.1.4 Impact on Plan Recognition Architecture

In the plan recognition algorithm of section 2.2, propositions describing the world state evaluate to true or false, the evaluation is known to be certain, and thus interpretations are either valid or invalid. With the introduction of TMS justifications, there is a middle ground between valid and not valid—described by degree of credibility. In this section, we discuss how a plan recognizer can capitalize on this new information, as well as deal with the fact that assumptions about the state of the world may be wrong.

Effective plan recognition involves making timely and knowledgeable choices between competing interpretations. Deferring decisions, thereby allowing further actions to narrow the field, restricts the ability to make inferences about the current situation. Making arbitrary choices is computationally expensive since they often have to be re-visited. In

[Kautz & Allen, 1986], the domain-independent assumption that the preferred interpretation contains a minimal number of top level plans is used to make reasoned choices in the presence of uncertainty. In [Carver et al., 1984], both domain-independent and domain-dependent heuristics could be expressed and used to guide choices. Credibility represents an additional source of discrimination knowledge, as described below.

3.1.4.1 Use of Credibility

As an example of the use of credibility, consider the following scenario. Let building of system version SV be in progress as described in Figure 3.5, where substantive changes were made during source editing, standard testing is assumed operative, cases TC1-TC3 are assumed applicable, and only case TC1 has been run. Further, let there be assumptions that new testcases are needed, and that archiving is not waived. The state of the plan for building SV is shown in Figure 3.7. Let the next action be *edit*. Given the operator library of Figure 3.3, there are two interpretations for *edit*: editing to make a new testcase and editing to get source code ready.

Consider the interpretation of *edit* as part of making new testcases; in this interpretation (Figure 3.8), *edit* continues the work of building system SV. *Edit* itself has no preconditions, but it inherits the precondition *newcases_needed* from *make_one_newcase*. Since *newcases_needed* is true by-assumption, this interpretation has high credibility—it depends on (one) extended state proposition that is assumed to be satisfied. Note that if *newcases_needed* had been true with-certainty, then the interpretation would be valid absolutely.

Now consider the interpretation of *edit* as part of getting source code ready; in this interpretation, *edit* starts a new top level plan (shown to the right in Figure 3.9). Again, *edit* has no preconditions, and *make_newmodule* has no preconditions, but *make_source* has a precondition that the baseline system (on which this new system version is to be based) is

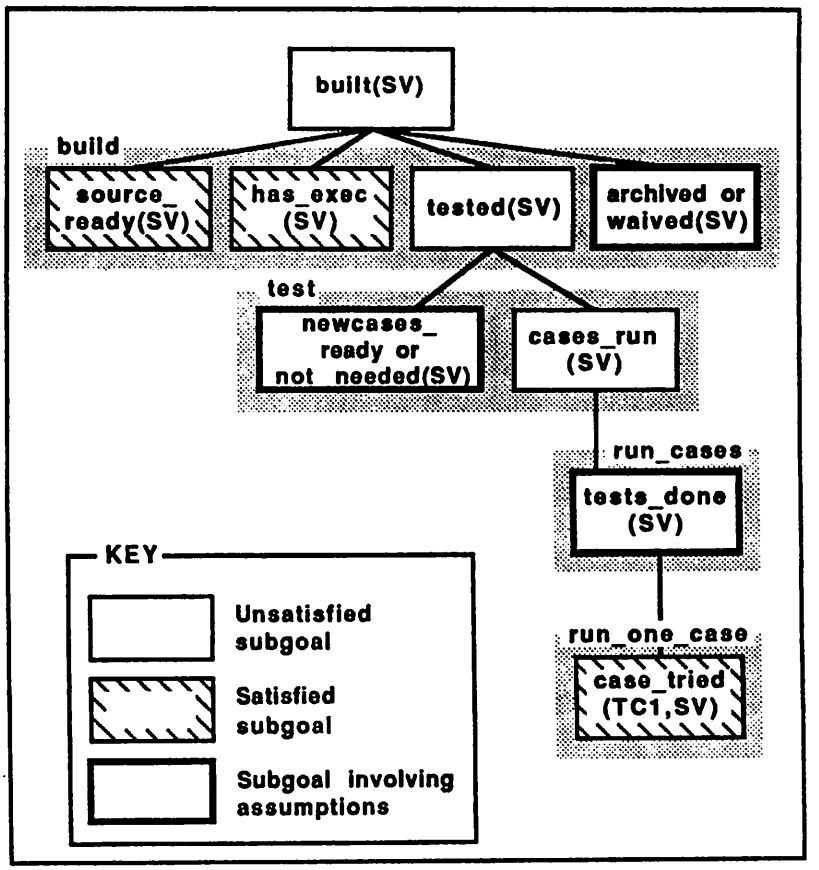

Figure 3.7 A Plan in Progress

built. *Built(SV)* is not true, but there is a plan to achieve it that is in progress. Three assumptions enter into believing that this plan is not finished (a discussion of how these assumptions are identified appears in section 4.1.2.2.2). They are that new testcases are needed (affecting the first subgoal of *test*), that testing is not done (affecting the iterated subgoal of *run_cases*), and that archiving is not waived (affecting the last subgoal of *build*). Thus, the interpretation of *edit* as part of starting a new *build* plan has low credibility— it conflicts with three current assumptions.

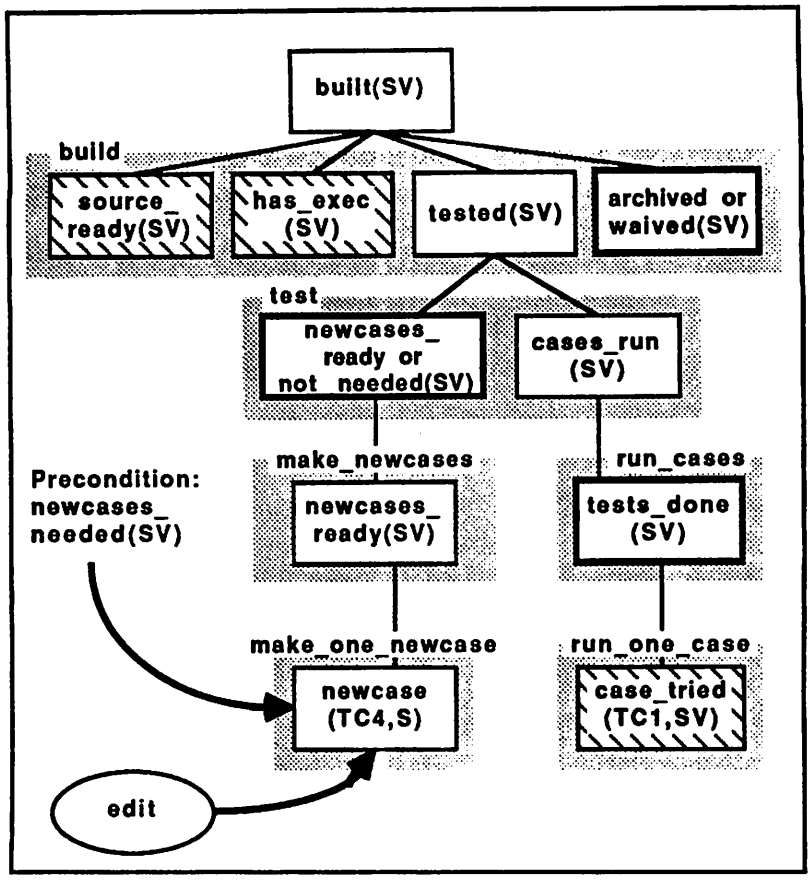

Figure 3.8 Edit as Making Testcases

Credibility is a basis for distinguishing the relative likelihood of these two competing interpretations, establishing a clear preference for *edit* as making testcases. Had it been the case that *newcases_needed* was false by-assumption, then the interpretation involving *edit* as making testcases would conflict with one current assumption. In this case, both interpretations violate current assumptions, so the user could be notified that the *edit* action is possibly in error and given the chance to reconsider performing that action. If the user chooses to perform the *edit*, then the interpretation of *edit* as making testcases is (still) the most credible interpretation.

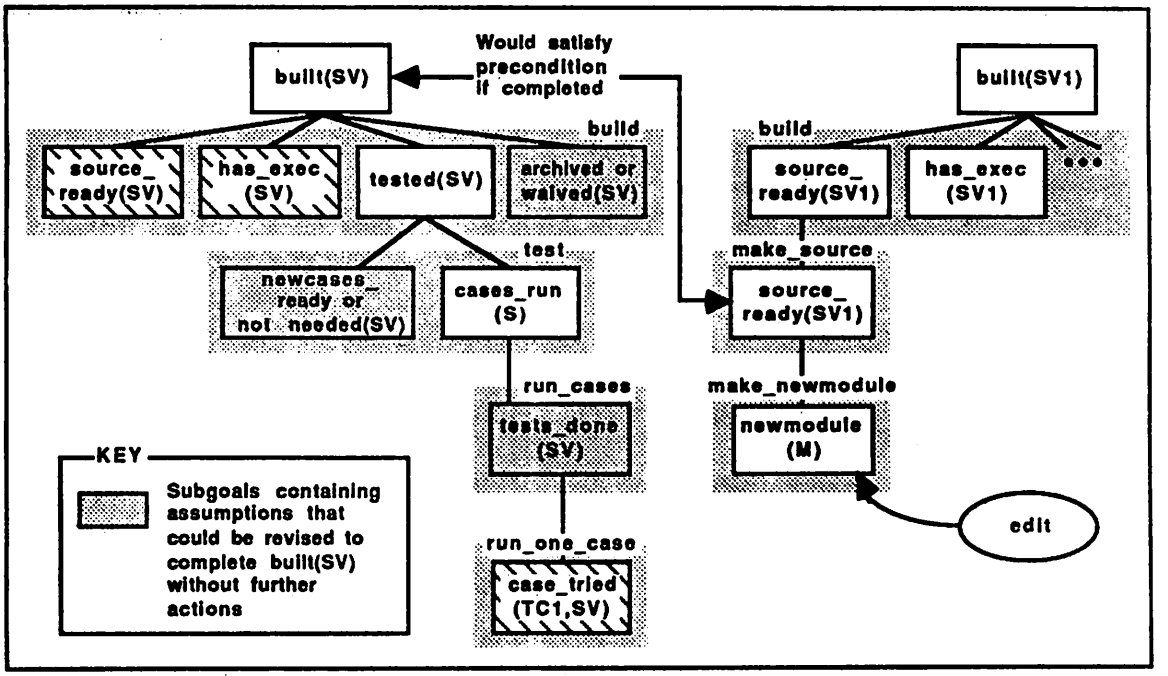

Figure 3.9 Edit as Preparing Source

It so happens that discrimination based on the domain-independent heuristic preferring interpretations with the minimal number of top level plans reinforces the preference for edit as making testcases, although in other cases there will be conflicts between credibility and the minimal-plans criterion. The best discrimination decisions will result from combining the evidence from multiple, independent perspectives. Credibility represents a new perspective, derived from deeper modeling of the context of actions.

3.1.4.2 Reconciliation

Reconciliation is the process of revising assumptions to make the world state conform to the requirements of an interpretation. This is only necessary when the "best" alternative (considering all available discriminators) still violates a few assumptions about the state of the world, or when other more attractive alternatives were originally chosen but were

subsequently disqualified. The standard approach to reconciliation would be to adopt the necessary assumptions (by giving them new justifications), and propagate the consequences through the TMS. This may lead to contradictions (a node and its negation both IN), which can then be resolved via dependency-directed backtracking [Stallman & Sussman, 1977] as implemented for TMS's [Doyle, 1979].

The standard approach to reconciliation misses an important opportunity—the opportunity to explain why the desired assumptions should hold. Each node N to be brought IN during reconciliation is simply provided with a new justification *without re-assessing why N failed to be IN* (i.e., without re-assessing any of the labels on nodes appearing in the currently invalid justifications for N.) This means that clues that other assumptions are wrong will be ignored. That is, since no attempt is made to bring N IN by making one of its currently invalid justifications valid, no assumptions in the foundations of N that might have been wrong will be revised.

As an example, consider how to explain that testing is in fact done given the situation diagrammed in Figure 3.5; this is one part of reconciling the interpretation shown in Figure 3.9. To bring *tests_done(SV)* IN, we must force *not tests_done(SV)* OUT (see justification J10); that means forcing OUT both the nodes *applicable(TC2,SV)* and *applicable(TC3,SV)* (covering both instances of justification J9). This can be done in two ways. One way (involving justifications J6 and J8) is to bring IN both *spec_excluded(TC2,SV)* and *spec_excluded(TC3,SV)* by adding two "dummy" justifications to support these nodes. The other way (involving justifications J1, J2, J4 and J5) is to block justification J1 that currently supports *standard_test(SV)*; note that this change affects only this instance of rule J1, not any other instances. This latter alternative is heuristically preferred, since it involves one rather than two changes. The result of choosing and installing this alternative is shown in Figure 3.10.

This approach to reconciliation attempts to achieve a more complete integration of new information into the TMS. The idea of finding explanations for desired beliefs is based on nonmonotonic reasoning in the following sense: if a proposition is found to be true, then *typically* one of existing justifications for the proposition should be valid. This type of reasoning is appropriate when all the possible reasons for a proposition being true are represented by justifications in the TMS, which is true for this application, but not for TMS applications in general. Reconciliation can be implemented by extending the algorithm for dependency-directed backtracking to include making invalid justifications valid as well as making valid justifications invalid. This algorithm for reconciliation will be described in detail in section 4.2.

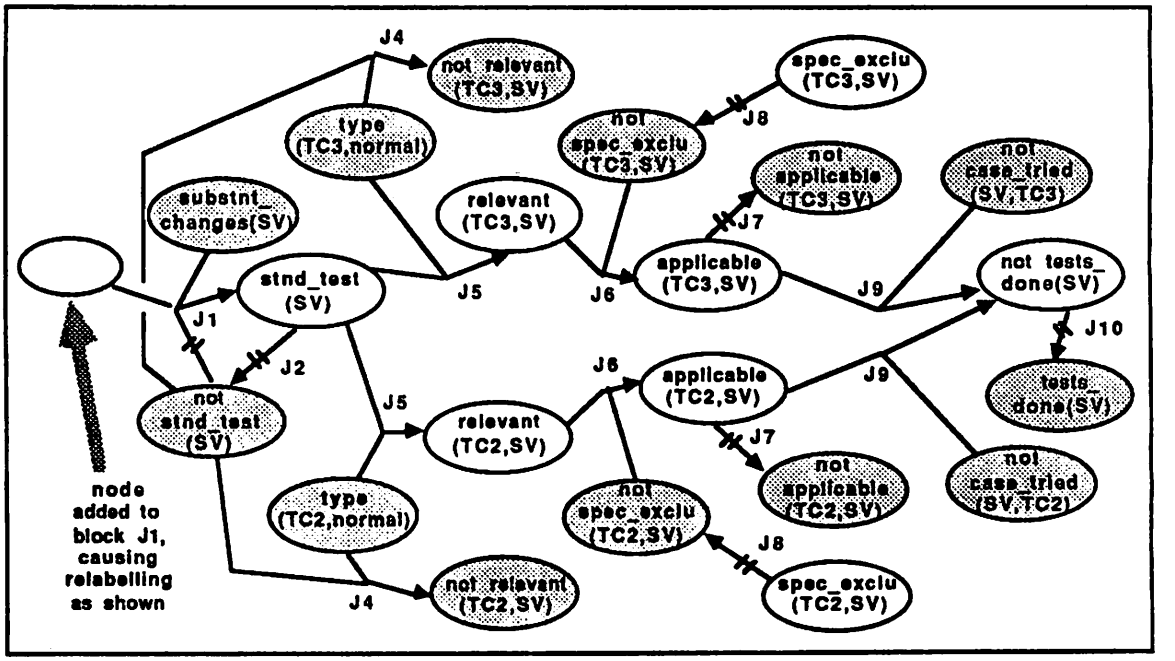

Figure 3.10 Example of Reconciliation

No matter what approach is used for reconciliation, it is possible for reconciliation to fail. This happens when an interpretation depends on assumptions that cannot hold in the current world state. In this event, the interpretation should be rejected.

3.1.5 Summary

We have shown that missing state information is a barrier to achieving substantive support in an intelligent assistant. Given that information is missing, the only way to exploit additional background knowledge about the context for actions is to introduce uncertainty, in the form of empirical knowledge that makes plausible assumptions about missing data. We have shown how to formalize this additional knowledge (theoretical and empirical) using a TMS to implement nonmonotonic reasoning. Then, we have shown how a plan recognizer is affected when the underlying state of the world includes both certainties and assumptions. Assumptions determine the credibility (rather than validity) of competing interpretations, and credibility can be used to make reasoned choices. We have also shown that the plan recognizer will sometimes find its assumptions to be faulty, and we have introduced a new method for revising assumptions in response to this.

In the remainder of this chapter, we consider additional examples of this technique (section 3.2), and then provide some guidance in writing justifications to capture domain knowledge (section 3.3).

3.2 Additional Examples

In this section, we present additional examples of this approach to deeper domain modeling. This section serves three purposes: to give further specific examples of justifications, to show some special techniques for improved operator definitions that the justifications make possible, and to show that there is a substantive body of domain

knowledge that can be captured in justification form. Finally, we discuss the role of justifications in the context of a knowledge-based approach to software products.

3.2.1 Releasing and Promoting System Versions

As new versions of a system are incrementally developed, selected versions are released to the customer. Customers expect bug-free releases with successively greater functionality. Thus, in order to be *releasable*, a version should have more functionality than the current release and should also not be buggy. On the other hand, it sometimes happens that a version is discovered to have serious bugs only after it has been released; then a commonly-used strategy to provide the customer with a working release promptly is to "re-release" the previous release. If a system version does not meet either of these criteria for releasability, then it should be considered *not releasable*. These three rules about releasability are shown in Figure 3.11 as rules J1-J3.

This reasoning about releasability cannot be exploited without a representation of the predicates *more-function* and *buggy*. Since these predicates are not directly observable, we need clues from other predicates. *More-function* is readily approximated by a development time stamp—versions developed later generally have more functionality than earlier versions. Bugginess can be approximated by looking at the officially reported bugs in the bug database—bugs (especially the more serious ones) are generally reported promptly and recorded there. The rules for determining *more-function* and *buggy* are shown as J4-J7 in Figure 3.11 (the preference rules will be introduced and discussed in section 3.3.2).

One example of how these rules work under reconciliation is as follows. Suppose there is no reason to believe that the current release C is *buggy*, but that the user takes an action to "re-release" the previous release R. The TMS will show that R is *not releasable*, so this action does not have the highest credibility. In order to treat such an action as if it were legal, we have to reconcile *releasable(R)*. R would be *releasable* by rule J2 if only C

J1	current-release(C) and more-function(R,C) EXCEPT buggy(R) or not releasable(R) —> releasable(R)
J2	current-release(C) and buggy(C) and previous-release(C,R) EXCEPT not releasable(R) —> releasable(R)
J3	EXCEPT releasable(R) -> not releasable(R)
J4	developed-after(R,C) EXCEPT not more-function(R,C) —> more-function(R,C)
J5	more-function(R,C) —> not more-function(C,R)
J6	EXCEPT more-function(R,C) —> not more-function(R,C)
J7	has-bug(S,B) and severity(B,high) EXCEPT not buggy(S) —> buggy(S)
J8	EXCEPT buggy(S) —> not buggy(S)
	PREFER releasable(R) PREFER buggy(S) PREFER more-function(R,C)

Figure 3.11 Justifications for Releasability

were *buggy*. By adopting this assumption, that C is *buggy*, we make R *releasable* and explain why that is so. The new information acquired, that C is *buggy*, can now be used to recognize further actions. For example, it would be needed to explain why more tests are run on C.

The justifications in Figure 3.11 follow the pattern of capturing both fundamental domain principles and empirical knowledge (as described in section 3.1.3.3). The empirical knowledge appears in the justifications that approximate the predicates *buggy* and *more-function*. The domain principles are evident in justifications J1-2 where the criteria for releasing are defined, and also in justification J6 which defines *more-function* as an anti-reflexive predicate.

Rules similar to those shown in Figure 3.11 can be used to determine the promotability, as opposed to releasability, of system versions. On multi-person projects, code developed by an individual is "promoted" to be visible to (and useable by) other project members. Just as for releasing, it is expected that successively promoted versions have successively more function and are not buggy. (The definition of *buggy* may be based on less stringent criteria than in the case of releasable, and therefore may have to be formalized as a different predicate—*too-buggy-to-promote*—but the ideas are the same.)

3.2.2 Inferring the Type of Contents of a File

In Figure 3.12, we give a set of justifications for inferring whether the contents in a file is source code or documentation. The qualifier used on the file name can be used as a clue to the *type*: special (system-recognized) qualifiers are usually used for source code (such as "C" code) and users often employ other special qualifiers for other entities (documentation, test cases, etc.) Another clue is the *type* of the *predecessor* to this contents (a *predecessor* relationship is set up between two items when one is edited to create the other). These clues lead to nonmonotonic justifications, because they are not infallible indicators of *type*. In addition, there are three monotonic rules in Figure 3.12. Rule J5 captures the knowledge that if the contents of a file has compiled (successfully), then it must contain code; the final two rules provide that documentation and code are mutually exclusive types.

While this example captures only empirical knowledge and thus does not really contribute to "deeper" domain modeling, it is a useful example for two other reasons. First, it shows that the truth value of an extended state predicate can sometimes be set entirely via monotonic rules; J5 is such a rule for *type*. So, it is possible for extended state predicates to be true or false *with-certainty*. Second, it shows that it is possible for rules to "conflict". For example, when a file has a code qualifier but its *predecessor* is of *type*

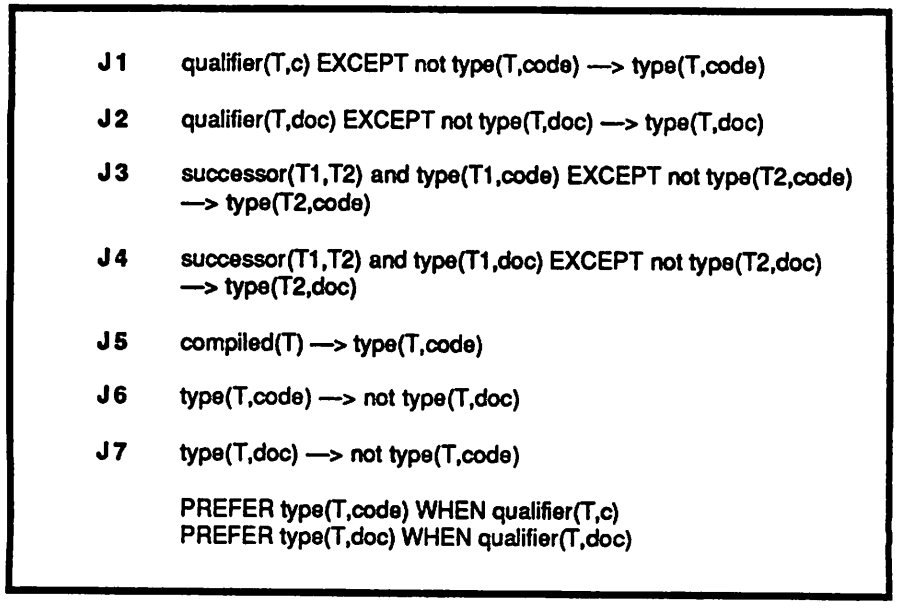

Figure 3.12 Justifications for Type

documentation, the two clues from which *type* is inferred support opposite conclusions. The handling of this problem is discussed in section 3.3.1.4.

3.2.3 Modeling Optional Activities

In writing the operators for software development, certain activities appear, on first analysis, to be optional, and therefore it is not clear just how to treat them in an operator definition. Two examples are as follows. During system build, some systems should be archived in the source code control facility, while for others this step is skipped. And, during testing, a user will sometimes browse through the source code of certain modules that were used to create the load module being tested.

An optional activity can always be represented trivially, by an iterated subgoal that is iterated(0), e.g., repeated zero or more times. This simple approach does not address the issues of when an optional activity ought to be performed, or what the implications of

performing the optional activity might be. Use of nonmonotonic reasoning makes this possible. First, justifications make it possible to define additional predicates that can be used to approximate when to skip an optional activity and when to perform it. Second, the very fact that an optional activity occurred may be used as evidence, via nonmonotonic justifications, that other propositions are now true in the world. We describe these two approaches below.

3.2.3.1 Performing versus Skipping Optional Activities

The handling of optional archiving during system build has already been shown in Figure 3.3. There, the *build* operator has a subgoal *archived(S) or archive_waived(S)*. This subgoal can be satisfied by the *archive* operator, or it can be satisfied if *archive-waived*, an extended state predicate, is true via some set of justifications. (Note that representing explicitly when to archive, via a static precondition on the *archive* operator, is not enough to solve the problem; in addition, it is necessary to represent how the archiving subgoal is satisfied when the *archive* operator is disallowed.) The justifications for *archive-waived* must approximate the conditions under which it is allowable to waive the archiving step. For example, one might take the view that archiving can be waived under two circumstances: if there are bugs in S that are to be fixed immediately, or if the predecessor to S was archived and the differences between S and its predecessor are cosmetic (affect comments, not executable source statements.)

3.2.3.2 Reasoning from Occurrences of Optional Activities

A different treatment is used to handle a user browsing through source code during testing. Here, the important issue is not allowing or disallowing this activity, but rather drawing the appropriate conclusions when it is done. This is because browsing the source code during testing is usually an indication that there is a bug newly revealed by one of the test cases just run. Thus, we treat browsing source code as an optional activity, using an

iterated subgoal in the *test* operator as shown in Figure 3.13. The operator which can achieve this subgoal is *browse-source*, which has *ADD code-examined(S)* among its effects. (Notice that *browse-source* is restricted, through the constraint, to just those source versions which made up the load module being tested.) The predicate *code-examined* can then used as evidence in a nonmonotonic justification to infer that there are suspected bugs in S. The predicate *suspected-bugs* may be used in other justifications, for example as part of inferring whether all the appropriate bug reports have been issued.

Running test cases on old system versions during preparation of source code for a new system build is another optional activity that can be treated like browsing through source

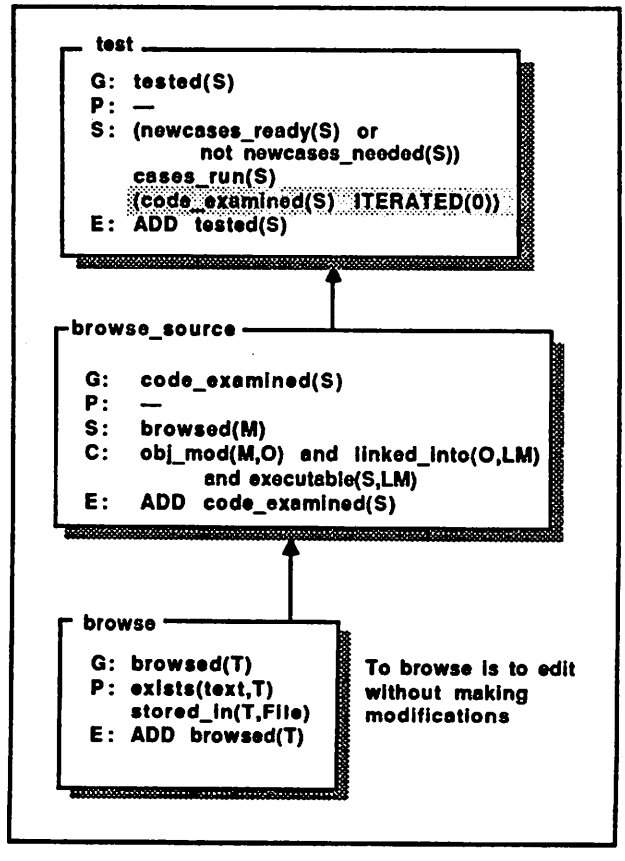

Figure 3.13 Browsing Source Code during Test

code during testing. If the test case is one that has a reported bug in it whose to-be-fixed-by date is near, then the running of the test case is typically evidence that the programmer is about to implement the fix (and happens to be running the test case in order to understand the bug or refresh his memory of it.) If the test case is a new one, just created, then running the test case may simply be part of understanding what features are already implemented, or how an existing feature works in a special case. Then, there is no connection with intent to fix a bug.

3.2.3.3 Optional Activities and Offline Operators

In section 2.1.2.6, we introduced the notion of offline operators to account for "actions" that do not take place on-line; an example was given where making a baseline choice (prior to building a new system version) was represented as an offline operator. We suggested that use of offline operators leads to better modeling of the process, especially in the case where there are other actions that could be taken in support of the offline operator. Now we are in a position to show how this can be done using the techniques for handling optional activities and nonmonotonic reasoning.

Consider the offline operator for making the baseline choice (this was given in Figure 2.7). It is possible that deciding what baseline to use may involve other actions that are aimed at understanding the characteristics of the candidates. These actions might include running test cases against existing versions or browsing through source code. However, it is not guaranteed that the programmer needs to do any of this—these supporting actions are optional.

This can be handled with the following approach. In the offline operator *choose_baseline*, we insert a core precondition *baseline_known(System)*. We provide a new operator *analyze-baseline-candidates* that achieves *baseline_known*. This new operator will be complex, with an ITERATED(0) subgoal that is satisfiable by the desired

set of information gathering actions. Now, if no activities are undertaken in support of choosing the baseline, *analyze-baseline-candidates* completes due to the ITERATED(0); if activities are undertaken, they are accounted for in the subgoal of *analyze-baseline-candidates*.

It is possible to expand this treatment by giving *analyze-baseline-candidates* a precondition that is an extended state predicate, and giving justifications on this predicate. Then, an interpretation involving *analyze-baseline-candidates* will be preferred when this extended state predicate is assumed true (i.e., when such analysis is expected), and will be downgraded in likelihood otherwise. (This is important because information gathering actions such as running test cases and looking through source code can have many possible interpretations, and therefore there is a need to distinguish between them.) For example, the most common baseline choice is the most recent system version, and no analysis is usually needed. However, when there are a vast number of new bugs in the most recent system version, the programmer may be considering whether to use an earlier version as the baseline, so that the buggy code can be reimplemented from scratch.

As our examples show, some kinds of optional activity are extremely common in the software development application, and therefore it is very important to be able to account for these actions in a meaningful way. Much of the actual on-line activity can be characterized as information-gathering—looking at test cases to see if they are applicable, running test cases to identify the cause of a bug, comparing two versions of source code to identify differences, etc. Systems like UNIX, which have a multitude of facilities for manipulating textual material, encourage these kinds of activities. In one set of UNIX terminal sessions that we studied, information-gathering accounted for nearly one third of all commands issued. Without the facilities for deeper domain modeling described in this chapter, the role of these activities cannot be represented except in a trivial way.

3.2.4 Test Case Applicability

The justifications given in Figure 3.4 through which test case applicability is determined are fairly simple-minded (from the software engineering perspective) and need to be made more realistic. In this section, we discuss the sources of knowledge through which this can be accomplished. Obviously different projects, and even different people on the same project, may take different approaches to the problem of how to do unit testing. Therefore, the following discussion is meant to give examples of the kind of reasoning that might be used, rather than the kind of reasoning that must be used.

There are multiple areas for improvement: inclusion of additional test strategies, more accurate determination of the operative test strategy, additional factors (other than test strategy) that affect applicability, determination of the category (base, normal) of a testcase, and determining when testing is done. The discussion below is limited to describing the type of testing usually called unit testing; there is no intention to cover here the other types of testing, such as regression testing, performance testing, capacity testing, etc.

3.2.4.1 Types of Testing Strategies

Weak testing (running the base cases only) and standard testing (running base and normal cases) are not the only testing strategies. Sometimes, testing can be skipped entirely; this could be called the no-test strategy. Such a strategy is acceptable if source changes affected only the comments in the source code. Another testing strategy, appropriate when lots of new features have just been implemented, can be called the new-features strategy. This strategy involves running the base cases and only the new normal cases (those especially created to cover these new features.) A final testing strategy example is the bugs-only strategy, used when the code was modified to fix one or more outstanding bugs. In this situation, only those normal cases that were affected by the bugs would be run, along with the base cases.

A slightly better approach is to separate the testing strategy for base cases from that of normal cases. For example, base cases might only be run when the editing changes to the source were either extensive or very limited. Running of normal cases would be determined independently, and might involve running all, some, or none of the normal cases as indicated above.

3.2.4.2 Operative Test Strategy

As indicated by the foregoing discussion, the operative testing strategy is often predictable from the types of editing changes made to the source code as part of constructing a new version. There are multiple dimensions along which these changes can be measured: what percentage of source code was changed, what percentage of the changes were additions and what percentage were modifications, were chunks of code deleted, were the changes localized or pervasive, were the changes limited to comments, etc. In addition, it is possible to measure the percentage of source modules changed, to take into account which ones were changed, and to determine if new modules were created.

There are other sources of clues about operative testing strategy as well. The outcome of testing of the previous version might indicate whether or not this version is aimed largely at fixing bugs or at adding new features, which in turn affects the use of bugs-only or new-features strategies. For example, if testing were prematurely halted on the previous version (see section 3.2.4.5), then the current version is probably aimed at fixing bugs. The relative importance of the bugs officially reported in the bug data base might also enter into determining if bugs are being fixed in this version. In addition, the type of testing applied to the previous version might affect the test strategy for its successor. For example, if the previous version were tested by the new-features strategy and there are only limited differences between that version and the current version, then a standard test strategy might

be called for. Certainly, the cumulative test coverage since the last archived version could also play a role.

3.2.4.3 Other Factors Affecting Testcase Applicability

The justification set in Figure 3.4 allows relevant test cases to be specifically excluded, and simply provides that typically a case is not specifically excluded. Test cases have to be excluded when they are affected by a catastrophic bug that renders the case useless; such a test case remains specifically excluded until an attempt is made to fix that bug. When a new bug is reported against a test case, and the bug report shows a scheduled to-be-fixed-by date that is considerably in the future, and the bug is related to the test case to blowing up, looping infinitely, etc., then a conclusion that this test case has become specifically excluded is warranted, at least until the to-be-fixed-by date.

In addition to specifically excluding test cases, it is also possible to specifically include them. The usual reason for specifically including a test case is that the test case has a bug in it that is supposed to be fixed in this system version. We have already seen (section 3.2.3.2) that running an existing test case (with a reported bug that is due-to-be-fixed) against an old system version during preparation of source code for a new build can be evidence of intent to fix a bug.

3.2.4.4 Categories of Test Cases

No justifications were provided in Figure 3.4 for determination of the category of a testcase. Two categories were presumed: base and normal cases. Base cases cover a representative set of features (the most commonly used features, or the most important ones); they tend to be used as "touchstones"—to determine if a new version is basically on track. Often, there is just a single base case. Normal cases are intended, collectively, to meet the needs of unit testing; at any given time, for the features that have been

implemented to date, the normal cases cover these features fairly exhaustively. (On larger projects, the normal cases may need to be thought of as divided up into application-specific categories, where a single category contains cases covering a set of related features; these categories will usually be reflected in the directory structure used to store the test cases.)

The best evidence for the category of a test case has to do with the directory it is stored in. Normal cases might be stored in a directory that contains only other test cases or perhaps miscellaneous text files. Base cases might be stored in a directory along with a load module.

3.2.4.5 Completion of Testing

Rules J9-10 in Figure 3.4 provide that testing is not complete until all applicable test cases are run. However, often unit testing stops as soon as a number of new bugs have been identified; either there is no purpose to running more cases until these bugs are fixed (because the bugs render the tests useless) or the identification of additional bugs is being deferred (since the programmer has plenty of work to do given the already identified bugs). To cover this case, another rule should be added deducing *testing-done* monotonically from *testing-short-circuited.* This requires additional rules for deducing, nonmonotonically, when *testing-short-circuited* is true. One such rule is that *testing-short-circuited* is true when a base test case has blown up.

3.2.4.6 Other Related Knowledge

The outcome of testing, i.e., the discovery of new bugs, has already been shown to be related to testing strategy. (Discovery of new bugs, as well as proof that old bugs are now fixed, also enters into determining if the bug report database is up-to-date. Bringing the bug database up-to-date, by reporting new bugs and closing out old bugs that are now

fixed or no longer duplicatable, can be used as an additional subgoal in the build operator.) We briefly consider how to reason about test outcomes.

Reasoning about discovery of new bugs is a two-part process. First, is there a bug? Second, is this a new bug? Rules to predict occurrences of bugs can be formulated relatively easily; we give four examples. If a test case blows up, then it probably has a bug (it might not if the blow up was due to unimplemented code scheduled to be written later). If a test case is run via a debugger, and the debugger is used to change data or code, then the case probably has a bug. If a test case is first run without the debugger and then run again via the debugger, then it probably has a bug. And, if the test output is significantly shorter than on the previous execution of this case (under the same conditions), then there is probably a bug.

Rules to predict whether a bug is a new bug are harder to formulate. There are two reasons this is hard. First, bugs can affect multiple test cases, not just the case the bug was originally discovered in (or originally reported against in the bug database). Second, a test case can have more than one bug. A simple approach to predicting new bugs shows the difficulties. Let one set of rules establish that generally bugs do not affect testcases other than those they are reported against. Another rule provides that if a bug occurs in a testcase and a reported bug that is not yet fixed affects that testcase, then the bug is not a new bug. A final rule provides that otherwise it is a new bug. The net result is that a bug in a testcase will be considered a new bug unless there is a bug already reported against that test case. This is weak reasoning, because new bugs will be masked by existing bugs in the same test case and reoccurrence of bugs across testcases is ruled out by default.

3.2.5 Use in Knowledge-based Product Context

As the foregoing discussion shows, there is ample opportunity to apply nonmonotonic reasoning to achieve deeper modeling of process activities. The use of nonmonotonic

reasoning is even more powerful when it is applied in the context of a knowledge-based approach to software products.

In the typical environment today (such as a generic UNIX environment), software products such as specifications, test cases, design documentation, and the like are treated largely as textual objects; source code is treated as a structured object during compilation, but there is still no deeper understanding of the purpose or role of individual code segments. As a result, much useful knowledge about these products is not explicitly recorded and therefore is not available to be used to support the programmer who is creating and maintaining those products. Knowledge-based product approaches, such as [Rich & Waters, 1988; Czuchry & Harris, 1988], are aimed at modeling the deeper structure of software products.

It might be thought that, in a knowledge-based product context, there would be little need for the nonmonotonic reasoning, since there is so much additional state information available. But this is definitely not the case. It may happen that some extended state predicates in the generic environment can be treated as core predicates in the knowledge-based-product environment, but three other phenomenon will offset this. First, the additional state information will make it possible to include new extended state predicates for which there would otherwise be no basis for predicting. Second, more accurate (but still imperfect) means of predicting some of the existing extended state predicates will be available using this new information, so some nonmonotonic rules are replaced by better rules that are still nonmonotonic. Finally, nonmonotonic reasoning may be used to arrive at some of the knowledge about the products; when these assumptions are in turn used to reason about process state, the conclusions are reached by-assumption, not with certainty.

3.3 Translating Domain Knowledge into Justifications

In this section, we give additional technical detail on the form and use of justifications. In the first subsection, we use an example to show the kinds of considerations that must be taken into account to ensure that the justifications correctly implement the desired domain knowledge. In the second subsection, we define an additional facility, preferences, that simplifies the way justifications are written in certain circumstances.

3.3.1 Writing Justifications

The translation of domain knowledge into various forms of nonmonotonic rules has been studied within the default logic framework [Reiter & Criscuolo, 1981]. These results can be applied to TMS justifications (always remembering the distinctions between the two approaches). Two forms of nonmonotonic rules have been identified: normal (A:MB/B) and seminormal (A:MB&C/B). Using the simple default logic to TMS translation, TMS normal form is A EXCEPT ¬B —> B and TMS seminormal form is A EXCEPT ¬B OR C —> B. All the examples presented in the literature fit one of these two forms. Indeed, it is convincingly argued in [Lukaszewicz, 1985] that the other rule forms are not well-founded. (His argument is along the following lines: Any rational agent should be willing to replace a rule A:MB/C with the more restrictive rule A:MB&C/C; if not, then the agent must be aware of a case where the first rule can be applied but the second cannot. In this case, it follows from the applicability of the first rule that A is provable and B is consistent; since the second rule is not applicable and we already know that A is provable, then B&C must not be consistent. The only way for B&C to be inconsistent when B is consistent is for C to be inconsistent. But in that case, the agent is trying to apply A:MB/C to conclude C *when C is inconsistent*—and that is not rational behavior.)

Knowing that justifications should be in normal or seminormal form still leaves open the question of exactly when to use one form in lieu of the other, and it does not give useful

guidance for writing justifications. We use an example to illustrate the many factors entering into the process of formalizing domain knowledge in justification form.

Consider the domain knowledge relating to the extended state predicate B. Suppose that there is one absolute piece of information: if A1 holds, then B is true; and further suppose that there are three approximate pieces of information: if A2 or A3 hold, then B is typically true while if A4 holds, B is typically false. Thus, A1 is certain evidence for B, A2 and A3 are probable indicators for B, and A4 is a probable indicator for ¬ B. There is a straight forward translation of this domain knowledge into four justifications (one monotonic, the rest nonmonotonic) as follows:

J1: A1 —> B
J2: A2 EXCEPT ¬B —> B
J3: A3 EXCEPT ¬B —> B
J4: A4 EXCEPT B —> ¬B

These justifications demonstrate a particular advantage of normal form: each piece of domain knowledge is written as a separate justification without consideration of the support conditions in the other justifications. For example, in J4, the use of EXCEPT B occurs in lieu of any mention of the negations of A1, A2 or A3, the support conditions for B appearing in the other justifications. This independence among justifications is a very desirable characteristic. For example, if another justification is later added for concluding B, the existing justifications concluding ¬B do not have to be rewritten.

This translation may or may not capture the desired domain knowledge, as we show below. Formalizing the domain knowledge in justifications may point to cases that were previously overlooked, or cases where the original domain knowledge was unintentionally ambiguous or contradictory.

3.3.1.1 Support versus Exception Nodes

First we clarify the differences between normal or seminormal forms. Given two justifications involving the same predicates, the question is what difference it makes if a given predicate is placed before the EXCEPT or its negation is placed after the EXCEPT. Consider two examples: one in normal form, A EXCEPT ¬B —> B, and its "equivalent" in seminormal form, EXCEPT ¬A OR ¬B —> B. If A is a core predicate, the two justifications are actually equivalent because if A is IN, then ¬A is guaranteed to be OUT, and vice versa. Since the exception nodes are meant to capture nonmonotonicity (reasoning in the absence of certain information), and since there is always complete information about the core nodes, the first form should be used. If A is an extended state predicate, then the two justifications produce the same results except in the case where A is unknown (or in the case where A is a contradiction, but this is ruled out as described below). If A is unknown, then both A and ¬A are OUT; hence the first justification would be invalid but the second would be valid. The choice of form is dependent on which case actually reflects the desired domain knowledge.

The usefulness of (indeed, the necessity for) seminormal form is covered extensively in [Reiter & Criscuolo, 1981]. A representative case calling for seminormal form, taken from that paper, is the following. When typically A's are B's (A EXCEPT ¬B —> B), and typically B's are C's (B EXCEPT ¬C —> C), the conclusion that a typical A is a C will be drawn. However, the domain knowledge may in fact indicate that A's are *not* typically C's. Then the justification that typically B's are C's needs to be rewritten to capture this exception. This is done using seminormal form: B EXCEPT A or ¬C —> C.

3.3.1.2 Coverage

For each set of justifications with a common conclusion (for our example this means B or its negation), coverage of all relevant cases must be considered. If there can be a state of

the world in which none of A1, A2, A3 or A4 are true, then in that state of the world, B will have a truth value of unknown. That is, there will be no support for either B or ¬B because none of the justifications will be valid—there is simply no evidence affecting B. This may be exactly what was intended, but it could result from an oversight in writing the justifications. If it is an oversight, then additional justifications should be added until all relevant cases are covered.

There are some types of justifications that are easy to overlook. If a particular predicate represents a relationship that is known to be transitive, then a justification *p(x,y) and p(y,z) —> p(x,z)* should be included. Other justifications are needed to represent commutativity, reflexivity, mutual exclusivity, and the like. It is important to include all such relevant justifications, particularly when they are monotonic.

3.3.1.3 Contradictions

Next, any possible contradictions in the justifications should be considered. A contradiction arises when two monotonic justifications supporting opposite conclusions are valid. This indicates an error in expressing domain knowledge in the justifications. For example, adding A5 —> ¬B will cause a contradiction if A1 and A5 are not mutually exclusive. That is, when both A1 and A5 are true, B will be IN and ¬B will also be IN. The solution to apply depends on what the domain knowledge really is: both justifications can be dropped, they can be reexpressed (for example, as A1 AND ¬ A5 —> B, A5 AND ¬ A1 —> ¬B, and possibly a new justification to cover the case for A1 AND A5), or they can be made nonmonotonic (but then see section 3.3.1.4 below).

Note that when all nonmonotonic justifications are in normal or seminormal form, the case of a contradiction (i.e., a node IN and its negation also IN) caused by two nonmonotonic justifications simply cannot occur. The essence of normal/seminormal form, the appearance of the negation of the conclusion as one of the exceptions, is a guard

to prevent this from happening. However, there is an alternate possibility relating to ambiguity, discussed below.

3.3.1.4 Ambiguities

It is possible for ambiguities to arise, creating a situation where it is unclear which among several nonmonotonic justifications are meant to apply. (Ambiguities cannot occur between monotonic and nonmonotonic justifications—a monotonic justification always overrides a nonmonotonic justification. So, if A1 and A4 both hold, there is no ambiguity; B will be IN, and J4 will therefore be invalid.)

A real example of ambiguity has already been given in section 3.2.2; when a file has a code qualifier but its predecessor is of type documentation, the two clues from which type is inferred support opposite conclusions. In the hypothetical example we are now considering (justifications J1-J4 given above), no ambiguities are possible if A4 is mutually exclusive with both A2 and A3. As long as this is true, there is no question about which nonmonotonic justification to apply—for each situation, there is a unique choice. However, if A4 and A2 (or A3) can both be true, the justifications may have to be modified to reflect accurately the domain knowledge that applies. (See also [Reiter & Criscuolo, 1981] for discussion of this problem.)

Suppose it is possible that A4 and A2 can be true simultaneously (as noted above, no ambiguity arises if A1 is also true). Certainly, the informal expression of the domain knowledge is ambiguous about this case. It is open to at least three possible interpretations—B unknown (the two pieces of knowledge defeat each other), B true (reasoning from J2 is more compelling), and B false (reasoning from J4 is more compelling).

What does the TMS do? Use of justifications in normal and seminormal form leads to loops in the TMS; these loops are even loops which are the cause of multiple labellings

[Charniak et al., 1980]. To illustrate this phenomenon, the justifications J2 and J4 are shown in Figure 3.14. The loop is the path of arrows from B to ¬B and back to B; it is an even loop because it involves an even number (≥2) of exception links. When A2 and A4 are both IN, this configuration of nodes can be labelled in two different ways, one shown on the left and one shown on the right. Note that there is no TMS labelling corresponding to the possible interpretation that the two justifications "defeat" each other. (While the TMS can implement this latter case, it cannot do so with the justifications as originally written.)

In the presence of multiple labellings, the operation of the TMS, as traditionally implemented, is well-defined. Whichever justification first becomes valid will prevent the other justification from being made valid when its support later comes IN. So, using the justifications as written, if A4 goes IN before A2, ¬B will go IN; when A2 later comes IN, that fact that ¬B is IN will prevent the justification from being valid.

This behavior introduces an element of randomness into the truth values returned by the TMS, and as such is not desirable in this application. The problem is not really in the

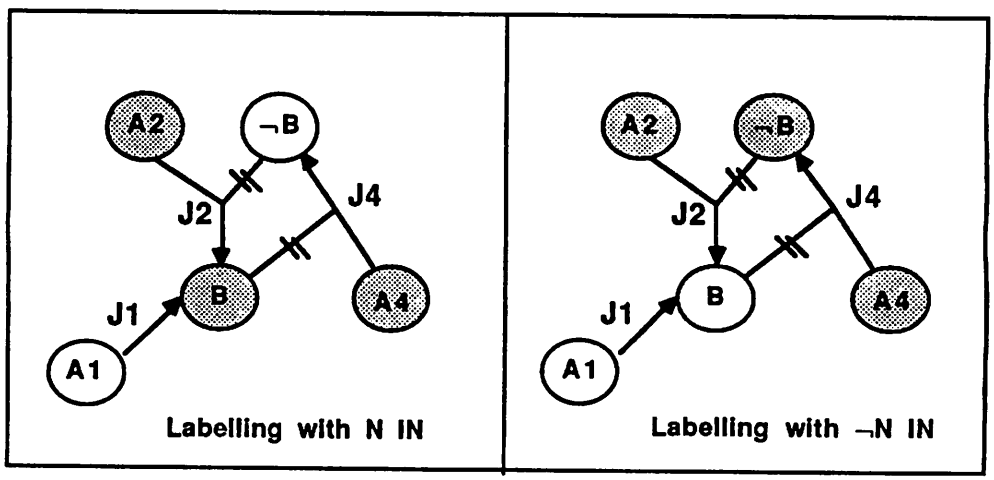

Figure 3.14 Even Loops in a TMS

TMS, but rather in the informal expression of the domain knowledge, which does not make clear what was actually intended in the A4 AND A2 situation. At a minimum, the justifications can be modified so that they read: A2 AND ¬A4 EXCEPT ¬B —> B and A4 AND ¬A2 EXCEPT B —> ¬B. This gives a truth value of unknown for B when A2 and A4 are both true. If there is domain knowledge to support the truth of B (or ¬B) when both A2 and A4 are true, an additional justification can be added.

This solution to achieving predicable TMS results has an unattractive side-effect: the independence of the justifications is affected. The justifications become somewhat tedious to write and maintain when cases must be disambiguated along more than two dimensions. In order to address this issue, we have experimented with a system of preferences as described in section 3.3.2.

3.3.1.5 "Otherwise" Cases

Sometimes, there is a need to express domain knowledge of the form " in those cases where A1, A2, ... or An are true, typically B holds and *otherwise*, typically ¬B holds". Then, a justification of the form ¬A1 AND ¬A2 AND... ¬An EXCEPT B —> ¬B must be used to express the "otherwise" case when ¬B holds. In particular, the use of a justification in the form EXCEPT B —> ¬B will not achieve the desired affect; this is another case of the even loop phenomenon. If none of the Ai is true initially, B will be OUT. The justification EXCEPT B —> ¬B will be valid, and ¬B will go IN. If one of the Ai subsequently becomes true, the fact that ¬B is IN will prevent any of the other justifications from being valid. Then the TMS labels do not reflect the "otherwise" aspect of the domain knowledge.

As with ambiguities, this type of justification destroys the independence among justifications. Therefore, it is worth considering how an "otherwise" justification can be conveniently written and interpreted. We argue that the only rational interpretation for a

justification of the form EXCEPT B —> ¬B, which has no support nodes and a single exception node, is that it is intended to be applied only as a last resort and overridden whenever possible. If that is not the intent then it should not be written so generally—additional conditions should be added to the left-hand side (either additional exception conditions or new support conditions).

However, the operation of the TMS is such that once such a justification becomes valid, it prevents any of the justifications supporting B from being made valid. The preference facility described in section 3.3.2 can be used to force this rational interpretation of EXCEPT B —>¬B on the TMS behavior.

3.3.1.6 Considering Time

A final consideration has to do with the behavior of the justifications over time. (A TMS does not make any distinction between belief revision and changes of state.) Since the TMS is being used to model successive snapshots of a changing world, we want to capitalize on the fact that conclusions vary over time as their supports and exceptions vary. If A goes IN and there is a justification of the form A —> B, then B goes IN; if, at a later point in time (i.e., in some subsequent state of the world) A goes OUT, B also goes OUT (assuming no other justification supporting B is valid then.) Ensuring that nodes go IN and OUT at the intended times requires being careful about the time-varying properties of the predicates on the left and right hand sides of the justifications.

As an example, consider two closely related core predicates with different time-varying qualities: *current-customer-release* and *customer-release*. *Current-customer-release* is true for only one system version at a time—that system version that is the release currently in use by the customer; *customer-release* is true for more and more versions as time goes by—any version for which *current-customer-release* ever became true. Assuming these are core predicates, then the *release* operator effects would include deleting *current-customer-release*

for the old release, adding *current-customer-release* for the new release, and also adding *customer-release* for the new release.

Use of one predicate or the other in a justification determines the time-varying characteristics of the conclusion in the justification. Some assumptions about a system version may be sanctioned only during the time that that system version is the current customer release; these assumptions are expressed in nonmonotonic justifications that include *current-customer-release* as one of the support nodes. Then, as soon as a new release is made, the assumptions about the old release will be retracted, which is the desired behavior. Other assumptions about a system version may be sanctioned if the system version was ever made visible (i.e., released) to the customer; these assumptions are expressed in nonmonotonic justifications that include *customer-release* as one of the support nodes. In this case, when a new release is made, assumptions about the old release are unaffected, again the desired behavior.

In this way, the justifications refer to time qualitatively (making implicit reference to the time interval during which one or a set of propositions is true) rather than quantitatively (using explicit timestamps or time intervals defined between timestamps). The onus is on the writer of the justifications to encode the time-varying properties correctly. There is no underlying model of time to simplify this task.

3.3.2 Preferences

As described in section 3.3.1.4, ambiguities in domain knowledge may become apparent when the knowledge is formalized into justifications. If not addressed, these ambiguities give rise, in a TMS, to even loops where multiple labellings are possible (under certain circumstances) and these multiple labellings are resolved randomly. As already shown, it is possible to rewrite the justifications to clarify the ambiguous cases (then, the circumstances causing multiple labellings never arise, although the even loops remain).

But, this approach involves careful crafting of justifications; the net result is a group of justifications that are not readily modifiable.

An alternative approach would provide for the representation of knowledge about ambiguities (and the justification interactions that cause them) separately from the justifications themselves. This would make justifications easier to write (and to read and understand), by removing the complexities relating to special conditions. We have experimented with a preference facility that covers one type of justification interaction: cases of ambiguity about which justification to apply. Preference rules are written that effectively define precedences of some justifications over others (in TMS terms, preference rules identify preferred labellings in even loops). (The term "preferences" is used in [Petrie, 1987] for rules that guide decisions made in dependency-directed backtracking in TMS's; our preference rules represent a different concept.)

3.3.2.1 An Example

Consider the justification example used previously:

J1: A1 —> B
J2: A2 EXCEPT ¬B —> B
J3: A3 EXCEPT ¬B —> B
J4: A4 EXCEPT B —> ¬B

For an ambiguity (i.e., a multiple labelling) to arise, it must be possible for A2 and A4 to be true simultaneously and/or for A3 and A4 to be true simultaneously. Suppose both cases are possible, and suppose the domain knowledge actually indicates that J4 should always prevail over J2 or J3. This is expressed as a single preference PREFER ¬B WHEN A4. The preference is interpreted as follows: when A4 holds, the preferred labelling in the loop is that B be OUT and ¬B be IN. This does not mean that when A4 holds, ¬B will be guaranteed; that would be equivalent to interpreting the preference as if it

were the monotonic justification A4 —> ¬B. Rather it means that when A4 holds, ¬B will hold if that is possible within the constraints of the other existing justifications; a monotonic justification can prevent the preference from being applicable. For example, if A1 also holds in addition to A4, the preference for ¬B cannot be acted upon because B is IN via the monotonic justification J1.

Some other examples are as follows. If J2 and J3 are meant to prevail over J4, then the preference is PREFER B WHEN A2 OR A3. In other cases two or more preferences may be required. The combination of PREFER B WHEN A2 AND A4 with PREFER ¬B WHEN A3 AND A4 AND ¬A2 could be used. Unconditional preference rules are also allowed, where the WHEN clause is not needed. Real examples of preference rules appear in Figures 3.11 and 3.12.

Preferences can also be used to enforce the intended meaning of "otherwise" justifications. Any appearance of a justification in the form EXCEPT ¬B —> B necessarily implies the (unconditional) preference rule PREFER ¬B. With this preference, the otherwise justification has lowest priority—it will be overridden by any justification concluding ¬B.

3.3.2.2 Preferences and Reconciliation

There is a special advantage to the use of preference rules, in lieu of rewriting the justifications, that becomes apparent when reconciliation is needed. (Refer to section 3.1.4.2 where reconciliation was introduced and an example given.) Use of preferences leads to more natural explanations being generated during reconciliation. Consider two alternative sets of justifications, shown below. On the left is the justification set we have been using (with an explicit preference added) and on the right is a variation in which the justifications have been rewritten to capture the preference information.

J1: A1 —> B	J11: A1 —> B
J2: A2 EXCEPT ¬B —> B	J12: A2 and ¬A4 EXCEPT ¬B —> B
J3: A3 EXCEPT ¬B —> B	J13: A3 and ¬A4 EXCEPT ¬B —> B
J4: A4 EXCEPT B —> ¬B	J14: A4 EXCEPT B —> ¬B
PREFER ¬B WHEN A4	

Suppose initially that A2 and A4 are true with certainty; then ¬B will be IN in both the left or right hand cases. Now suppose that the assumption about ¬B becomes untenable, and reconciliation must be invoked to make B IN (and ¬B OUT). For the justification set on the left with an explicit preference rule, all that is needed is to block J4; then *J2 is* automatically made valid. In the case on the right, J14 must be blocked, but an additional justification must also be created to support B—J12 cannot be applied because it mentions ¬A4. Because of the way that justifications had to be rewritten, the most natural explanation, that A4 implies ¬B doesn't in fact override A2 implies B, is not possible; instead, it is necessary to create a justification that says "B is IN, but we don't know why".

3.3.2.3 Extending the Preference Concept

The preference rule facility should be regarded as a first step towards a general system for representing knowledge about justification interactions. Although the preference rules are useful, especially in the case of "otherwise" justifications which turn out to be very common, there are interactions they cannot handle. For example, another type of rule is needed to "cancel out" competing justifications; preferences can only select among competing justifications. Thus, if the intention is that no conclusion be drawn about B when A2 and A4 are both true, the preferences cannot accomplish this—the justifications must be rewritten accordingly (as A2 AND ¬A4 EXCEPT ¬B —> B with A4 AND ¬A2 EXCEPT B —> ¬B). In addition, the initial implementation of preference rules (described in Appendix D) was designed with clarity and correctness in mind, and is not very efficient.

3.4 Summary

In this chapter we have described an approach to deeper domain modeling. Missing state information was identified as a barrier to achieving substantive support in an intelligent assistant. Given that there is hidden state, the only way to exploit additional background knowledge about the context for actions is to introduce uncertainty, in the form of empirical knowledge that makes plausible assumptions about missing data. We have shown how to formalize this additional knowledge (theoretical and empirical) using a TMS to implement nonmonotonic reasoning. We have also shown how a plan recognizer is affected when the underlying state of the world includes both certainties and assumptions; this involved the two concepts of credibility and reconciliation. Additional examples of this technique were given, and guidance in writing justifications to capture domain knowledge was provided.

The implementation details of this approach to deeper domain modeling are presented in the next chapter.

CHAPTER 4
IMPLEMENTATION OF DEEPER DOMAIN MODELING

In this chapter, we supply the implementation details for our approach to deeper domain modeling. The chapter is divided into two major topics. Section 4.1 gives the technical and algorithmic details of marrying nonmonotonic reasoning (about the state of the world) with a plan recognizer. This includes the TMS implementation, changes and additions to the basic plan recognition algorithm, the calculation and use of credibility, and the interface to reconciliation. Sections 4.2 and 4.3 present reconciliation, a new approach to revising assumptions in a TMS. The motivation for a new approach is given, the approach is described, and then all the technical issues are enumerated and resolved.

4.1 Detailed Aspects of the Implementation

In this section, we describe a revised plan recognition architecture that accommodates the deeper domain modeling. This architecture is given schematically in Figure 4.1, which is a version of Figure 2.9 enhanced to highlight the additions required to implement the deeper domain modeling.

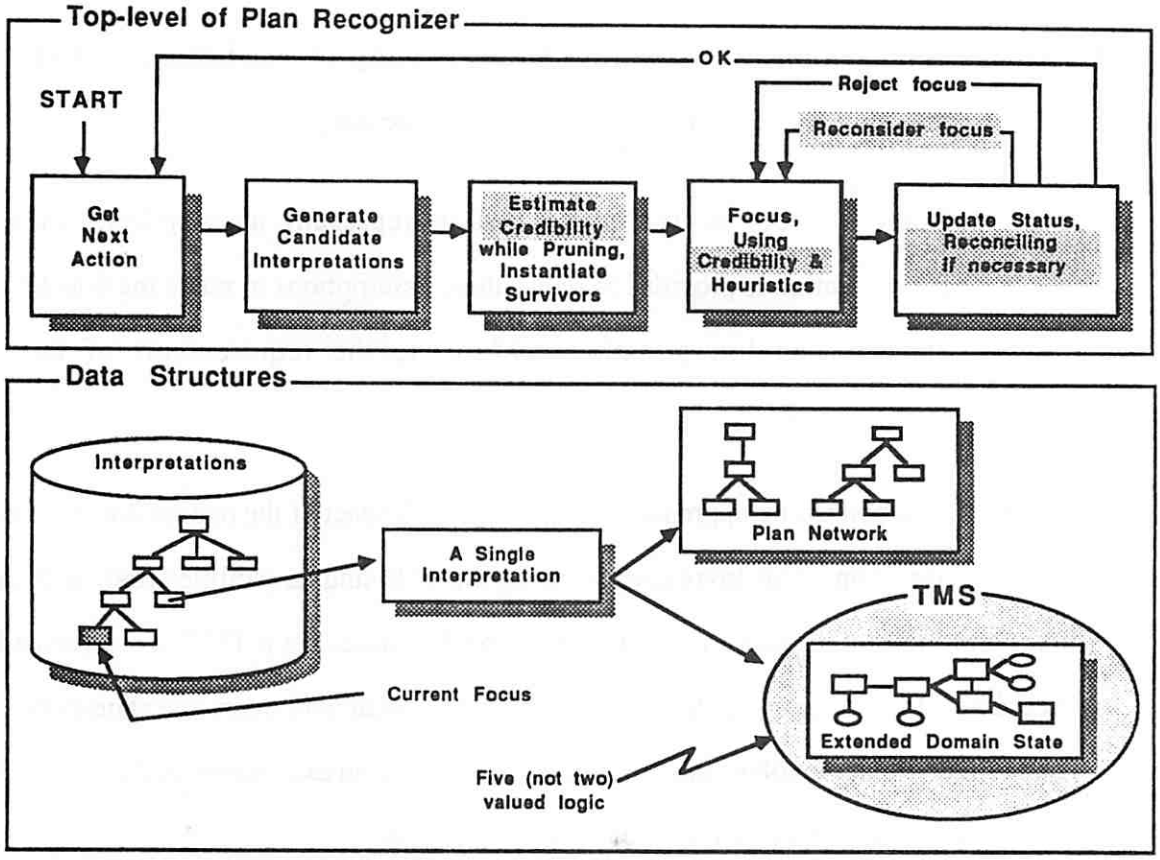

Figure 4.1 Revised Architecture for Plan Recognition

The major changes are these:

- The domain state is encapsulated in a TMS, not represented directly, and the TMS is used to represent additional domain predicates that were not previously available. All updating of the domain state and querying of the domain state must be via the TMS interface.

- Propositions about the domain state are represented in five, not two, valued logic. This affects expression evaluation during pruning of candidates and updating of status. In particular, it allows for the

computation of credibility during pruning of candidates, and the subsequent use of credibility rankings in focusing.

- Some aspects of the domain state represent assumptions, and reconciliation is provided to revise these assumptions to make the domain state for an interpretation conform to the requirements of that interpretation.

We take a bottom-up approach to describing the impact of the deeper domain modeling on plan recognition. This involves describing the TMS and its facilities first; after that, the direct and indirect impact on plan recognition of introducing a TMS are described. The direct impact involves using the TMS to represent, update and query the state of the world. The indirect impact involves the use of five-valued logic in expression evaluation within the plan recognizer and the possible need for reconciliation.

4.1.1 The TMS and Its Facilities

The basic TMS supports three operations in its external interface: adding a justification, deleting a justification, and returning the truth status of a proposition. Implicit in adding/deleting justifications are operations for creating new nodes in the TMS and for propagating changes to labels on nodes in accordance with the changes in the justifications; node creation and labelling are operations internal to the TMS, not callable directly by the problem solver using the TMS. In addition to the three basic externally-visible operations, the TMS implemented for use by the plan recognizer also has externally-visible operations to install preferences (as described in section 3.3.2) and to determine the certainty of labels on nodes. The internal algorithms and data structures used to implement this external interface are described in Appendix D. (Reconciliation is discussed separately in section 4.2.)

The present implementation of the TMS differs somewhat from other implementations reported in the literature (the original implementation in [Doyle, 1979] and later implementations in [Charniak et al., 1980; Goodwin, 1982]). In these implementations there are features not needed in the present application, and also desirable features that are missing. If we take Doyle's system as a baseline, then the significant differences in the present implementation, which is fully described in Appendix D, are as follows:

- Only the SL (support list) form of justification is used.
- The method described in [Rich, 1986] of indexing nodes by their roles in justifications is used to perform relabelling efficiently.
- An explicit distinction is made between node labels that are certain and those that are by-assumption. Nodes whose labels are certain either follow from premises and monotonic justifications involving only other nodes that are certain, or represent the negations of nodes that are certain. Those nodes that are certain cannot have their labels changed during reconciliation. When the TMS is queried for the truth status of a proposition about the domain state, the certainty is joined to the truth value to give a five-valued logic: certainly true, assumed true, unknown (implicitly assumed unknown), assumed false, and certainly false.
- A list is maintained enumerating **all** valid justifications supporting a given node, although only one valid justification is actually needed to label a node IN. Keeping complete lists of valid justifications is useful in distinguishing which node labels are certain; if only one valid justification is maintained for any IN node, it is possible to be misled into deciding that a given node is not certain (because its support structure involves nonmonotonic rules) when in fact it should be certain because

there is additional support that is strictly monotonic. Maintaining complete lists of valid justifications also facilitates analysis during reconciliation.

- Anomaly checks, to detect odd loops and potentially circular reasoning, are performed when a justification is instantiated. Odd loops correspond to pathological arguments where X IN leads to making X OUT which leads to making X IN, ad infinitum. Circular reasoning arises from variations on the basic theme of "X implies Y implies X"; these are even loops with zero exception links. Even loops with two or more exception links are allowed, as discussed in section 3.3.1.4. (The current GRAPPLE implementation rejects justifications that could cause circular arguments; an alternative approach is to allow such justifications, but check during labelling that no node is IN due solely to a circular argument.)

- An additional facility of preference rules is used to resolve interactions among rules that would otherwise lead to randomly selected labellings.

- Reconciliation, a variation on dependency-directed backtracking for revising assumptions, is used in lieu of dependency-directed backtracking.

- In this application, there is a class of nodes whose truth values are always either certainly true or certainly false; these are nodes representing core state propositions. An efficiency gain, allowing instantiation of fewer nodes, is possible if these nodes are treated under a modified closed world assumption. (This is explained in detail in Appendix D.) No special treatment applies to nodes for extended state propositions.

Before leaving the topic of TMS's, we mention the relationship of the TMS to a similar facility, the assumption-based truth maintenance system (ATMS) [deKleer, 1986]. This is relevant since the ATMS was introduced in response to certain identified problems in the TMS. When a TMS is used to implement search, a single solution is developed until a contradiction arises, at which point backtracking occurs to revise assumptions, leading to a new solution with which to continue the search. When two solutions have common subsolutions, work on the common subsolution may have to be repeated. The ATMS was designed to allow multiple solutions to be pursued simultaneously, thus facilitating the comparison of alternatives as well as the sharing of common results.

However, choice of a TMS versus an ATMS must be fitted to the problem at hand; neither is uniformly a better technique. As noted in [deKleer & Williams, 1986], there are problems for which TMS's (using a single context with backtracking) lead to more efficient search than ATMS's (using multiple contexts with sharing of common subsolutions).

The relevant issue here is the relative ability of the two systems to implement nonmonotonic reasoning. On this issue, the TMS is a more powerful system; the ATMS lacks a general facility for representing the nonmonotonic justifications for assumptions. Various extensions to the ATMS have been proposed to remedy this [deKleer, 1988], but none as yet provide the representational power of a TMS.

4.1.2 Use of TMS by Plan Recognizer

Having described both the external interface provided by the TMS (and alluded to its implementation described in Appendix D), we now turn to considering how the plan recognizer makes use of the interface. The plan recognizer invokes the TMS interface for three purposes: representation of states of the world, evaluation of expressions about the world, and reconciliation of world state. In addition, the plan recognizer must deal with the notion of credibility that arises from the introduction of five-valued logic through the TMS.

The relationship between the plan recognizer and the TMS is shown in Figure 4.2, and it is this relationship that is described in detail in this section. (The numbers on the boxes in the interface will be used to key the discussion to the relevant parts of the figure.) The revised plan recognition algorithm, showing the changes made to accommodate the deeper domain modeling generally and the TMS specifically, is given in Appendix E.

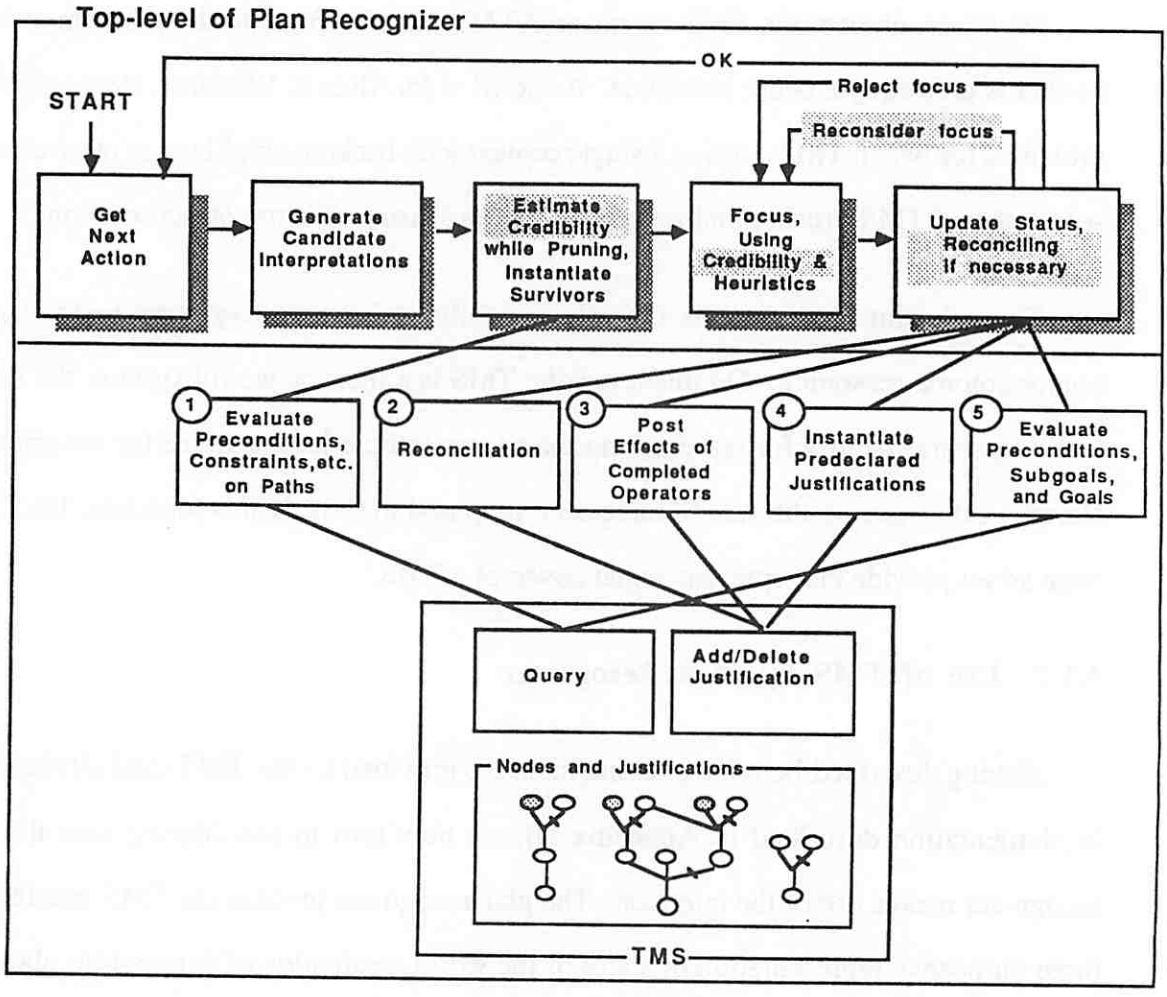

Figure 4.2 Interface between Plan Recognizer and TMS

4.1.2.1 Representing States of the World

In the baseline algorithm for plan recognition (discussed in section 2.2 and summarized in Appendix B), a very simple representation of the state of the world suffices. A list of the propositions that are true, together with the closed world assumption that anything not explicitly represented as true is false, is sufficient. (In fact, this representation was the original motivation for the use of the words ADD and DELETE in effects—to make a proposition true, it is ADDed to the list, and to make it false, it is DELETEd from the list).

In the remainder of this section, we discuss boxes 3 and 4 of the interface shown in Figure 4.2.

4.1.2.1.1 State in the TMS

In the revised plan recognizer, all state information is represented in the TMS, whose nodes represent both core and extended state propositions. ADD and DELETE effects are no longer implemented by simple list operations, but rather by the manipulation (addition, deletion) of premise justifications in the TMS. (Recall that NEW/REMOVE effects translate into ADD/DELETE operations on the predefined predicate *exists*). The implications of these premises are propagated through the network of pre-declared justifications that have been instantiated to arrive at the state information covering the extended state propositions.

Adding and deleting premise justifications, to implement effects clauses (box 3 of Figure 4.2), affects only the core predicates. In order to use the TMS to represent the extended state predicates, it is necessary to instantiate the domain-specific justifications declared with the operator definitions (box 4 of Figure 4.2).

Instantiation of pre-declared justifications involves binding parameters in the justifications to real objects in the domain world. It is clear that as long as domain objects

are created dynamically, instantiation must be an on-going process. It is insufficient to instantiate justifications for those objects that exist when recognition starts—every time a new object is created (via a NEW effect), new justifications need to be instantiated. The new justifications include those involving just the new object, as well as those involving relations between the new object and existing objects.

The algorithm in Appendix E shows this approach. New justifications are instantiated during the updating of the status of a plan network following processing on the path for the new action (but before pending preconditions and subgoals are processed). After these new justifications are instantiated, the node certainties are recomputed.

4.1.2.1.2 Extensions

We briefly mention two possible changes in the way that justifications are instantiated; these changes affect when box 4 of Figure 4.2 is invoked.

The two operations of instantiating justifications and computing node certainties, should, strictly speaking, be repeated after processing the pending preconditions and subgoals. Processing these pending conditions can lead to new effects being posted if new operators are found to have completed. If these new effects include creating new domain objects, there could be new justifications to instantiate for these objects; and, new premises have the potential to change the certainty of propositions. The algorithm in Appendix E is correct only when no complex operators have NEW effects, and when no predicates added/deleted by complex operators appear in justifications. Otherwise, instantiation of justifications and computation of node certainties should be repeated after each cycle of processing the pending conditions, until quiescence is reached.

(If node certainties are going to be computed multiple times, then it would be desirable to compute them incrementally as nodes are relabelled, rather than all at once. It is possible

to fold the determination of node certainties into the relabelling algorithm, so that the certainty of a node is established when the node label is established. This is somewhat tricky, as the computation must take into account the fact that the certainty of a node can be affected even when the node label itself is not changing.)

We have described an approach to instantiating justifications that is triggered on NEW effects (those creating new domain objects). An alternative approach, that is more efficient, is to trigger the instantiation of justifications on changes to core nodes via ADD/DELETE effects. When a justification involves a core node as one of its support nodes, and that core node is OUT with-certainty, the justification will be invalid. Further, it cannot be made valid (even by reconciliation) until premises are changed to make that core node IN, e.g., until an ADD affecting that core node occurs. The presence in the TMS of a justification with one or more core nodes OUT with certainty has two negative effects: it slows the labelling process and requires additional storage space.

An improved approach to instantiating justifications is realized by instantiating justifications on ADD effects and deleting already instantiated justifications on DELETE effects. When an ADD causes the last core support node in a justification to be IN with certainty, the justification is instantiated, independent of the status of any extended state predicates in the justification. When a DELETE causes any core support node in a justification to go OUT with certainty, then that justification can be deleted, independent of the status of any extended state predicates in the justification. A justification that has been instantiated and then deleted may later be instantiated again if all its core support nodes go back IN with certainty.

4.1.2.2 Evaluating Expressions about the World

There are two times when expressions about the state of the world need to be evaluated: when the possible interpretations for a new action are being generated (through

box 1 of Figure 4.2), and when the state of completion of plans in progress is being updated after execution of an action (through box 5 of Figure 4.2). In the baseline plan recognition algorithm, evaluation of a proposition about the state of the world was a simple matter. It entailed one of two answers: true or false. All state information was not only complete, but certain.

With the introduction of the default knowledge, evaluation of an expression about the state of the world entails a five-valued logic: certainly true, assumed true, unknown, assumed false, and certainly false. These are the possible results of querying the TMS for the truth of any proposition, as described in Appendix D. The truth of an expression, composed of these propositions and the logical connectives AND, OR, and NOT, is determined using truth tables for this five-valued logic. These truth tables are shown in Table 4.1. Using these truth tables, the truth value of an expression appearing in any clause (as a goal, precondition, subgoal, or constraint) of an operator definition can be evaluated.

The use of five-valued logic affects both how interpretations are evaluated and how the state of a plan is updated after an action is taken.

4.1.2.2.1 Evaluating Interpretations

When a path is evaluated to see if it can extend a given interpretation to provide a rationale for the next action, the relevant preconditions and constraints are evaluated along with the conditions "goal not achieved" and "top level goal not duplicated", as described in section 2.2.2.2. This is the work associated with box 1 of Figure 4.2. These path conditions used to evaluate to true or false, but now give one of five results. Since all these conditions must hold for the path to be valid, the validity of the path is obtained by and'ing together the results from each condition evaluation (using the truth table for AND in Table 4.1).

Table 4.1 Truth Tables

AND	certainly true	assumed true	unknown	assumed false	certainly false
certainly true	certainly true	assumed true	unknown	assumed false	certainly false
assumed true	assumed true	assumed true	unknown	assumed false	certainly false
unknown	unknown	unknown	unknown	assumed false	certainly false
assumed false	assumed false	assumed false	assumed false	assumed false	certainly false
certainly false	certainly false	certainly false	certainly false	certainly false	certainly false

OR	certainly true	assumed true	unknown	assumed false	certainly false
certainly true	certainly true	certainly true	certainly true	certainly true	certainly true
assumed true	certainly true	assumed true	assumed true	assumed true	assumed true
unknown	certainly true	assumed true	unknown	unknown	unknown
assumed false	certainly true	assumed true	unknown	assumed false	assumed false
certainly false	certainly true	assumed true	unknown	assumed false	certainly false

NOT	
certainly true	certainly false
assumed true	assumed false
unknown	unknown
assumed false	assumed true
certainly false	certainly true

If the path conditions evaluate to certainly false, the path can be rejected outright. If the result is certainly true or assumed true, consideration of the path can proceed without the need for reconciliation. If the result is unknown or assumed false, reconciliation will be needed before further work is done on the path. The results of evaluating the path conditions are also used to determine the path credibility, as described in section 4.1.3.1 below, which is used in turn to select the "best" interpretation to pursue first.

4.1.2.2.2 Updating Plan State

After an action has been interpreted and then executed, the state of the world is updated and then the state of the plan network is updated (as originally described in section 2.2.2.4). These activities are associated with boxes 3, 4 and 5 of Figure 4.2. The effects of the current action are posted, along with the effects of any newly-completed operator appearing on the path from the current action to the top (posting effects is represented in box 3, subsequent instantiation of justifications by box 4). Then, all pending preconditions and subgoals are evaluated (box 5) to see if any are now true, thus showing new parts of the plan completed as side-effects of the last action. When pending precondition and subgoals are found to have been satisfied, secondary updating may be possible. Pending preconditions that are newly satisfied may lead to the instantiation of the subgoals; pending subgoals that are newly satisfied may lead to the completion of an operator and the posting of its effects (boxes 3 and 4 again).

Whereas the evaluation of pending preconditions and subgoals (box 5) used to produce true/false answers, we must now contend with five possible answers. When a pending precondition or subgoal evaluates to certainly true, then the precondition/subgoal can be handled just as the "true" cases were handled in the two-valued case: the precondition/subgoal is marked satisfied, and secondary updating may follow. When a pending precondition or subgoal evaluates to certainly false, then the *precondition/subgoal*

can be handled just as the "false" cases were handled in the two-valued case: it is not marked satisfied, and no secondary updating will follow. In both cases, there is no doubt that the correct updates have been made.

However, when the answer is one of assumed true, unknown, or assumed false, there is room for doubt—the condition may or may not actually be satisfied in spite of the current evaluation. For these cases, preconditions and subgoals are handled differently.

When a precondition evaluates to assumed true, unknown, or assumed false, it is not marked as completed. However, if it is the case that all preconditions for an operator O are either already marked as completed or currently evaluate to one of these three values, then secondary updating is allowed—the subgoals of the operator are instantiated. If secondary updating is not performed in this circumstance, then some paths will be improperly excluded when the subsequent actions are processed. It is perfectly safe to instantiate subgoals before the preconditions are all true because the preconditions will be checked when a later action has a possible path leading to that subgoal. If at this time, any of these preconditions evaluates to unknown or assumed false, reconciliation will have to be called then.

Handling subgoals that evaluate to assumed true, unknown, or assumed false is done through an extension to the least commitment strategy described in section 2.2.2.5. We adopt a wait-and-see attitude—handling these cases as if the subgoal S were not satisfied. This means that no secondary updating will follow, even if the subgoal currently evaluates to plausibly true; so, the operator O owning S will not be marked as completed and its effects will not be posted.

Eventually, some action will be taken whose preconditions P require the effects of this operator. Then, the preconditions of this action will fail, and the planner will be invoked. The planner will find that completion of operator O will satisfy P, and that completion of O

depends on subgoal S being satisfied. If S then evaluates to assumed true, the planner can directly proceed to mark S satisfied, mark O completed, and post the effects of O. If S then evaluates to unknown or assumed false, the planner must first call reconciliation to make S assumed true, after which it can mark S satisfied, mark O completed, and post the effects of O. In either case, P will be satisfied (unless reconciliation has to be called and it fails, in which case the planner must report failure unless it can find another plan to satisfy P).

4.1.3 Credibility

The results of checking a path that is being considered as a way of extending the current interpretation to accommodate a rationale for the next action, by evaluating the preconditions, constraints, etc. as described in sections 2.2.2.2 and 4.1.2.2.1, are used to estimate the credibility of that path. In this section, we show how the credibility is determined, and then how it is used by the plan recognizer in focusing. The computation of credibility occurs as a by-product of the expression evaluation represented in box 1 in Figure 4.2.

4.1.3.1 Evaluating Credibility

The basic approach to evaluating credibility is simply to count the number of conditions (checks for preconditions and constraints) that evaluate to unknown or assumed false. Each of these cases represents a lack of support for the condition in the TMS, which reflects the judgment of the justifications. The higher the count of unknown/assumed false conditions, the lower the credibility of the path. (Note that when a precondition evaluates to certainly false, the planner is called; when a constraint evaluates to certainly false, the path can be rejected outright; and, cases where the evaluation result is certainly true or assumed true are cases where the requirements of the path are met in the TMS.) The example presented in section 3.1.4.1 showed this evaluation of credibility on two possible interpretations for an edit action. (In the case of the interpretation in Figure 3.9, some of

the violated preconditions/constraints were identified during planning to achieve a path precondition that originally evaluated to certainly false.)

This basic approach can be refined by making finer distinctions based on the results of the condition evaluations. A path with a condition that evaluates to unknown would seem to be preferable to a path with a condition that evaluates to assumed false, all other things being equal. In one case, the TMS is simply agnostic about the condition, but in the other case, the TMS actively supports the opposite view. Similarly, a path with a condition that evaluates to assumed true would seem to be preferable to a path with a condition that evaluates to unknown, all other things being equal. Finally, one could take the view that a path with a condition that evaluates to certainly true is preferable to a path with a condition that evaluates to assumed true. This view is perhaps more controversial, but it basically involves preferring something that is certain to something that is merely a possibility.

It is actually a simplification to base the credibility on the number of unsatisfied conditions (with or without considering the extent to which these conditions stray from being true.) Counting the condition violations ignores the possibility that two conditions fail to be true for the same reason. If one condition is A AND B and another is B AND C, and if A and C are both assumed true but B is merely unknown, then both conditions evaluate to unknown—and the effects of B will be counted twice. While it is unlikely that this kind of duplication would occur within the conditions of a single operator, the conditions that need to be evaluated on a path are collected from a set of operators, and duplications between conditions in two operators is a distinct possibility.

However, decomposing condition expressions into their component propositions still doesn't resolve the issue. Suppose the reasoning represented in the TMS shows both A and C follow monotonically from an intermediate proposition D that is assumed true. In

this case, looking at the propositions visible in the expression in the operator clauses is insufficient—D would still get counted twice instead of once.

The net result of this discussion is that credibility measures are approximations. An approximation is available rather cheaply, and computing a better measure would be rather expensive. (In fact, improved measures are not available until after reconciliation has actually been performed.) Since credibility is used as a heuristic, the extra expense of computing a more exact measure seems unjustified.

4.1.3.2 Using Credibility

The credibility of competing interpretations is used during focusing to rate the interpretations as to their relative likelihood of being the "right" interpretation. The idea is to pick the candidate with the highest credibility, and to pursue that candidate before doing any work on the competing candidates of lower credibility. Later, it may happen that new information, such as a subsequent action, reveals that what originally appeared to be the most credible alternative is, in fact, an impossibility. Then, the most credible of the remaining candidates is the next choice.

In the general case, credibility is a way of partitioning alternative interpretations into classes, where each member of a given class has the same credibility; there is no guarantee that each class will have a single member. When the most credible class of interpretations contains two or more alternatives, the other heuristics for focusing, described in section 2.2.2.3, should be brought in, to do the additional discrimination. If these heuristics still do not indicate a unique choice, a random selection among the top candidates will have to be made. We take the view that credibility should be used as the primary discriminator, with the other heuristics brought in as secondary discriminators if needed, because credibility is derived from a deeper model of the domain while the other heuristics are more involved with surface issues.

4.1.3.3 Extensions to the Use of Credibility

We briefly mention two additional ways that credibility can be used. These are not currently implemented in GRAPPLE.

There is an additional application for the notion of credibility, derived from a different measurement. Each interpretation in the entire interpretation tree (see Figure 2.8) has associated with it a state of the world (now stored in a particular state of a TMS). Each TMS contains some number of "patches" made to the justifications during previous invocations of reconciliation. A count of the patches can be thought of as a measure of the divergence of the actual state of the world from an "ideal" state determined by the normally applicable default knowledge. The higher the count, the less "credible" the interpretation. Comparing interpretations against this measure of credibility is expensive—it requires that they all be instantiated and that reconciliation be invoked if needed. But, it might be possible to define a "credibility threshold" such that an interpretation dropping below this threshold should temporarily be set aside (not actually rejected), and other alternatives considered.

Use of a credibility threshold would prevent the following undesirable situation. An initial decision to prefer interpretation A to interpretation B is made based on credibility and/or other heuristics. But then, interpretation of the subsequent actions frequently involves interpretations that are at best unknown or assumed false in credibility. Thus the plan recognizer keeps taking steps further and further "out on a limb", as indicated by the continual need to call reconciliation, when in fact it should be re-visiting the original decision to prefer A to B. Use of a credibility threshold would signal when the plan recognizer should consider other choices (although it will not pinpoint when the first wrong step may have been taken.)

Another refinement to the notion of credibility is possible. In the standard approach, credibility is estimated, interpretations are compared on that basis, and work then proceeds on the best choice. Among the very first things done to this choice is to call reconciliation, if needed. The impact of reconciliation can be measured by counting the difference in number of patches before and after reconciliation. If this number is significantly greater than the count of unknown and assumed false conditions (on which the original estimate of credibility was based), then the focusing decision could be revisited. For example, it might be thought advisable to pursue the next best alternative at least as far as determining the impact of reconciliation in that interpretation; then a more informed choice could be made.

4.1.4 Interface to Reconciliation

When the plan recognizer is faced with multiple alternative interpretations, those interpretations that do not require reconciliation are pursued first. When all such alternatives have been exhausted, reconciliation on the most credible of the remaining alternatives is required before that alternative can be pursued. Successful reconciliation guarantees that the current state of the world (core and extended state) is consistent with the requirements of that alternative. Reconciliation is represented by box 2 in Figure 4.2.

4.1.4.1 Calling Reconciliation

When reconciliation is called, the plan recognizer must identify all the assumptions that the validity of a path depends upon. This includes those conditions that evaluated to assumed true (even though they don't need to be reconciled) as well as those that are unknown or assumed false. In finding a reconciliation, the propositions that were already assumed true must be preserved in that state, while the others are made to be assumed true. If those that are already assumed true are ignored, then reconciliation could well settle upon a revision to the TMS that does not in fact support the path.

When reconciliation fails, it is an indication that the interpretation being considered depends upon a set of beliefs that are not consistent with the current state of the world and/or with each other. In this event, the interpretation should be rejected, and processing continued by re-focusing.

The reconciliation algorithm is, in the general case, faced with several alternative ways to reconcile an interpretation with the TMS; heuristics are used to pick a preferred solution. Therefore, it is possible for reconciliation to pick the "wrong" solution; this will affect the credibility of later actions, with the possibility that some actions appear to have lower credibility than they should. From one point of view, this situation presents no (new) problems. The TMS, at all times, reflects the state of the world according to the best information at hand; it is never guaranteed that this state is completely accurate. Propositions adopted through reconciliation may affect credibility of later actions, and may have to be "reconciled" again to pursue those actions. The current GRAPPLE implementation makes no attempt to distinguish between a case where the wrong choice was made in an earlier call to reconciliation and a case where the consequences of a choice made in an earlier call to reconciliation are simply no longer operative. This is an area for further investigation, as there is certainly some potential for capitalizing on this distinction.

Reconciliation, to explain action A, needs to be performed on the state of the world SW that holds just before A is executed. That is, the propositions that the interpretation of A depends on must be adopted then, but strictly speaking, the "reasons" for these propositions could be supplied later, perhaps after more information is available. This is best shown with an example.

Return to the example of section 3.1.3.1 with its justifications for test case applicability (Figure 3.4). Recall that applicability was a function of the operative test strategy and whether the particular case was specifically excluded. Suppose that in addition

to allowing testcases to be specifically excluded, it was also possible to specifically include them. Let the expected test strategy be weak testing, and let the first testcase T to be run be a standard test. There are two explanations for this action: that this testcase is being specifically included or that the test strategy is not weak but standard testing. To allow this action, all that is strictly necessary is to force IN *applicable(T)*; the explanation could wait. Choosing an explanation now will result in a random choice, if there is no basis to prefer one to the other. However, more information will be available as additional test cases are run. If further standard testcases are run, that would buttress the explanation that the operative test strategy is standard testing.

Delaying explanations, to achieve better explanations, is not at all a simple matter. There is, first, the question of how long to delay—when is there no longer the possibility or likelihood that new information would affect the explanation? Second there is the issue of timely feedback about other propositions whose status in the TMS is wrong; delaying explanation will delay this feedback. Finally, there is the issue of dealing properly with time, e.g., with multiple states of the world. Is the information gained in a later state of the world actually relevant to an explanation in the earlier state of the world? What is the affect on any intermediate states? More importantly, is there any affect on the interpretation decisions that were made based on those intermediate states? When reasons are adopted immediately as is done in the present algorithm, there is no need to save previous states of the world or to deal with more than the current state of the world at any time.

4.1.4.2 Considering Time

Some special considerations apply to propositions that do not vary over time; the truth values of such propositions cannot be changed arbitrarily to suit the convenience of the plan recognizer (i.e., during reconciliation). Suppose, as described in section 3.2.1, we had an extended state predicate *more-function(x,y)* which is true if load module x has more

implemented features than load module y (*more-function* ignores the issue of whether those additional features actually work or are *buggy*.) For a given x and y, the truth value of *more-function* does not vary over time. If *more-function(x1,y1)* is true at time t1, then it is never the case that *more-function(x1,y1)* is false at some time before or after t1. Therefore, if the plan recognizer is pursuing an interpretation in which the validity of action A1 depends on *more-function(x1,y1)* being true, it cannot later incorporate into this same interpretation a rationale for a later action A2 that depends on *more-function(x1,y1)* being false.

As explained in section 3.3.1.6, the original justifications are written to reflect the proper time variances. The issue being raised here has to do with constraining reconciliation so that these time variances are preserved. For example, suppose that *more-function(x1,y1)* had to be forced IN in order to interpret action A1; suppose this was done directly with a new dummy justification. There is nothing to prevent a later call to reconciliation, for action A2, from simply deleting this justification. In fact, we want this attempt at reconciliation to fail.

In order to handle this problem, it is necessary, first, to know which (extended state) predicates are not time varying. This has to be provided as additional domain knowledge, and can easily be done as part of the predicate definition (which already does things like enumerating the types of the arguments to the predicate, as shown in Appendix A). Then, whenever a non-time-varying proposition is actually used in an interpretation, its current truth value needs to be "locked" in. (Notice that we allow values to fluctuate freely, according to whatever the most informed view is at any given time, until they are actually used.)

Locking is done by a procedure that takes the same argument list as reconciliation: a list of nodes that need to be IN for this interpretation (call it I). A non-time-varying node N

then we explore some extensions and variations on this approach.

that appears on this list is "locked" IN by providing it with a premise justification. This premise justification captures the fact that if I is a correct interpretation, N must be true. (Since interpretations competing with I have their own states represented in separate

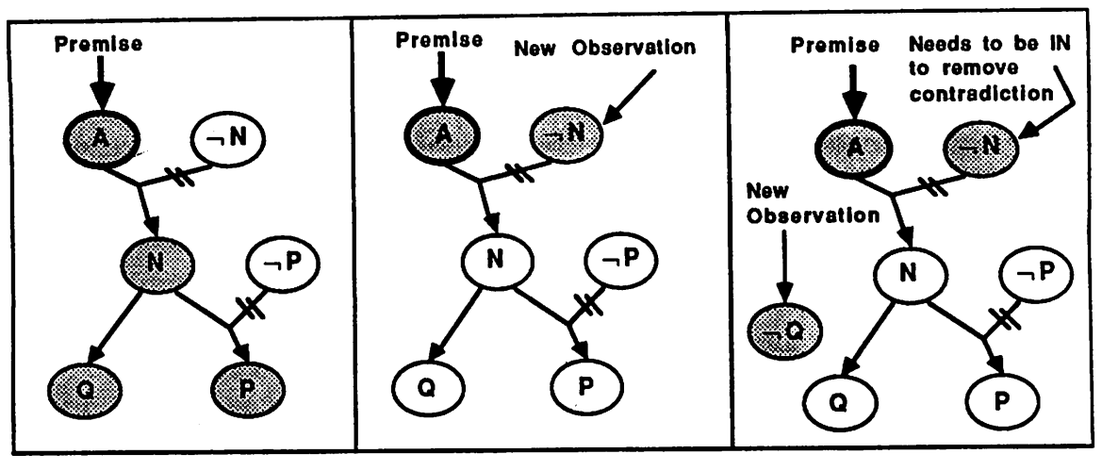

Figure 4.3 Incorporating New Information into the TMS

4.2.1 Reconciliation as Dependency-directed Backtracking

In a TMS, updating the labels on propositions to accommodate new information is handled by two mechanisms: normal relabelling and dependency-directed backtracking (DDB). Suppose the problem solver using the TMS learns that P is true although it had previously been assuming ¬P. The TMS is given a new justification to support P, and relabelling will occur (since P was not already IN). This relabelling ensures that the TMS reflects all the consequences of P being true. One of those consequences might be R; if it is the case that ¬R was already IN, then a contradiction will result. DDB is called to revise the labels so that there is no longer a contradiction. It does so by finding a "culprit": a node X that is IN via a nonmonotonic justification J, and that contributes to one of the contradiction nodes being IN. Then, it makes X OUT by making one of the exception nodes in J IN. In the example to the right in Figure 4.3, N is the culprit for the contradiction in Q, and the way to make N OUT is to make ¬N IN with a new justification.

implemented features than load module y (*more-function* ignores the issue of whether those additional features actually work or are *buggy*.) For a given x and y, the truth value of *more-function* does not vary over time. If *more-function(x1,y1)* is true at time t1, then it is never the case that *more-function(x1,y1)* is false at some time before or after t1. Therefore, if the plan recognizer is pursuing an interpretation in which the validity of action A1 depends on *more-function(x1,y1)* being true, it cannot later incorporate into this same interpretation a rationale for a later action A2 that depends on *more-function(x1,y1)* being false.

As explained in section 3.3.1.6, the original justifications are written to reflect the proper time variances. The issue being raised here has to do with constraining reconciliation so that these time variances are preserved. For example, suppose that *more-function(x1,y1)* had to be forced IN in order to interpret action A1; suppose this was done directly with a new dummy justification. There is nothing to prevent a later call to reconciliation, for action A2, from simply deleting this justification. In fact, we want this attempt at reconciliation to fail.

In order to handle this problem, it is necessary, first, to know which (extended state) predicates are not time varying. This has to be provided as additional domain knowledge, and can easily be done as part of the predicate definition (which already does things like enumerating the types of the arguments to the predicate, as shown in Appendix A). Then, whenever a non-time-varying proposition is actually used in an interpretation, its current truth value needs to be "locked" in. (Notice that we allow values to fluctuate freely, according to whatever the most informed view is at any given time, until they are actually used.)

Locking is done by a procedure that takes the same argument list as reconciliation: a list of nodes that need to be IN for this interpretation (call it I). A non-time-varying node N

that appears on this list is "locked" IN by providing it with a premise justification. This premise justification captures the fact that if I is a correct interpretation, N must be true. (Since interpretations competing with I have their own states represented in separate TMS's, there is no need to have a node in I's TMS explicitly representing the assumption that I is the correct interpretation.) If it happens that the next action requires ¬N among its preconditions (or constraints, etc.), there can be no rationale for that action as an extension to I. If it happens that reconciliation of the next action can only succeed if ¬N is true, then reconciliation must, and will, fail. Either of these two cases may eventually lead to rejecting interpretation I (since the correctness of I entails N and a contradiction is reached through N, I must not be a correct interpretation).

But, what about other nodes that cause these nodes to be IN—should they be locked too? For example, suppose there is a non-time varying proposition P that is an essential part of the argument that causes N to be IN. Perhaps, P should be locked (if P has to be OUT for N to be IN, then ¬P would be given a premise justification.) The trouble is that locking P may be incorrect: N may indeed be IN, but for a reason different than that shown by the TMS; then, future changes to P have been excluded improperly. The present implementation is overzealous: it locks P. A better solution is to put P on a list of nodes to avoid changing in reconciliation, and to use that list to select among competing reconciliation solutions.

An example where locking is needed arises in the rules for releasability, as shown in Figure 3.11. There, when a release is made under normal circumstances (newly released version V2 assumed to have more function than the current release V1 through reasoning on the development time stamp), rules J1 and J4 are valid. Since *releasable(V2)* is based on *more-function(V2,V1)*, which is not time-varying, *more-function(V2,V1)* is locked IN. Now, if a later action re-releases V1, there are two possible explanations: either V2 is buggy (rule J2) or V1 has more function that V2 (rule J1). The second explanation is ruled

out because *more-function(V2,V1)* is certainly true (due to locking) and therefore *more-function(V1,V2)* is certainly false from rule J8.

The notion of locking provides some control over time variance of propositions, but is not a general solution to preserving time varying properties in the presence of reconciliation. There can be other types of constraints on how the truth of propositions changes over time. As an example, consider a constraint of "once f(x,y) becomes true, it never becomes false"; this allows a one-time only transition from false to true for a given x and y. These types of constraints have been studied in the context of database integrity issues, which may suggest further avenues to explore.

4.2 Approaches to Reconciliation

Reconciliation is the process of incorporating new information into the current state of the TMS. This problem is exactly what nonmonotonic logics were specifically designed to handle. Given an initial set of observations and a set of rules (monotonic and nonmonotonic), a set of "facts" is derived. When a new observation is made, this set of facts may have to be revised. A simple example is given in Figure 4.3. The original observation of { A } is shown to the left; in this case, the derived facts are { N, Q, P }. If a new observation {¬N } is added as shown in the middle, then there is no longer any support for P or Q. On the other hand, if the new observation were ¬Q as shown on the right, then a contradiction arises because Q is already IN. This contradiction is resolved by supporting ¬N (which makes N OUT and therefore Q OUT); also, since N is no longer IN, P no longer has any support.

First, we describe the standard TMS approach to the problem of reconciliation, and then we explore some extensions and variations on this approach.

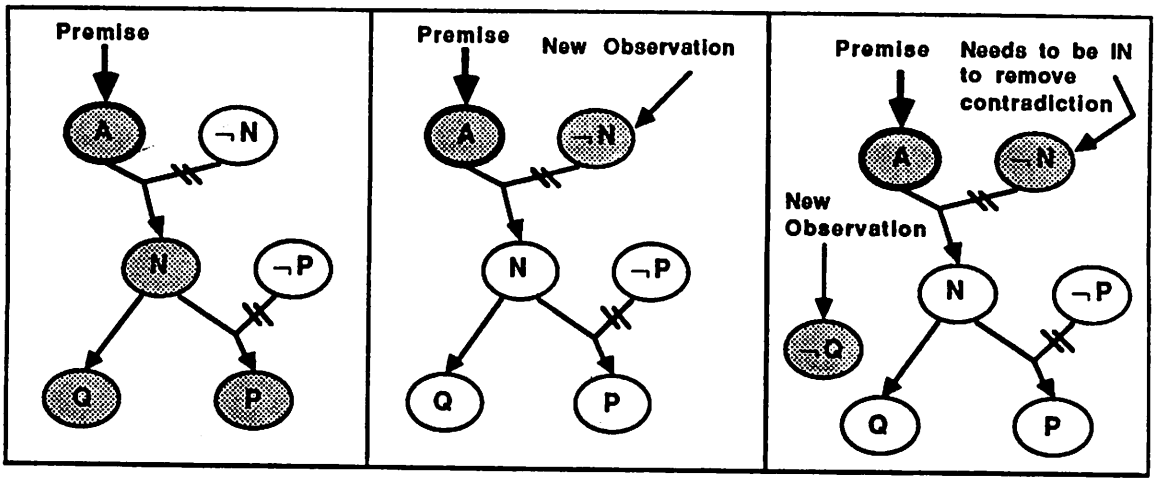

Figure 4.3 Incorporating New Information into the TMS

4.2.1 Reconciliation as Dependency-directed Backtracking

In a TMS, updating the labels on propositions to accommodate new information is handled by two mechanisms: normal relabelling and dependency-directed backtracking (DDB). Suppose the problem solver using the TMS learns that P is true although it had previously been assuming ¬P. The TMS is given a new justification to support P, and relabelling will occur (since P was not already IN). This relabelling ensures that the TMS reflects all the consequences of P being true. One of those consequences might be R; if it is the case that ¬R was already IN, then a contradiction will result. DDB is called to revise the labels so that there is no longer a contradiction. It does so by finding a "culprit": a node X that is IN via a nonmonotonic justification J, and that contributes to one of the contradiction nodes being IN. Then, it makes X OUT by making one of the exception nodes in J IN. In the example to the right in Figure 4.3, N is the culprit for the contradiction in Q, and the way to make N OUT is to make ¬N IN with a new justification.

Thus, one approach to reconciliation is to adopt this standard method for TMS's. New justifications are created for each node that should be IN (the form that these justifications should take is discussed in section 4.3.1.3). Relabelling follows, and contradictions may result. Then, DDB is called to resolve each contradiction. This algorithm for reconciliation is shown in Figure 4.4.

As described in [Doyle, 1979], DDB can be realized very efficiently using the TMS as part of the problem solving process: the algorithm for DDB is itself an application of nonmonotonic reasoning. That is, when faced with several culprits to choose from or several ways to make a single culprit OUT, DDB arbitrarily makes a choice and records this choice in the TMS nonmonotonically (e.g., in such a way that the alternatives will be revisited if the original choice is later found to be the wrong choice). The original implementation [Doyle, 1979] used a special form of justification for this; an improved technique appears in [Petrie, 1987]. DDB stops at the first complete "solution" it arrives at

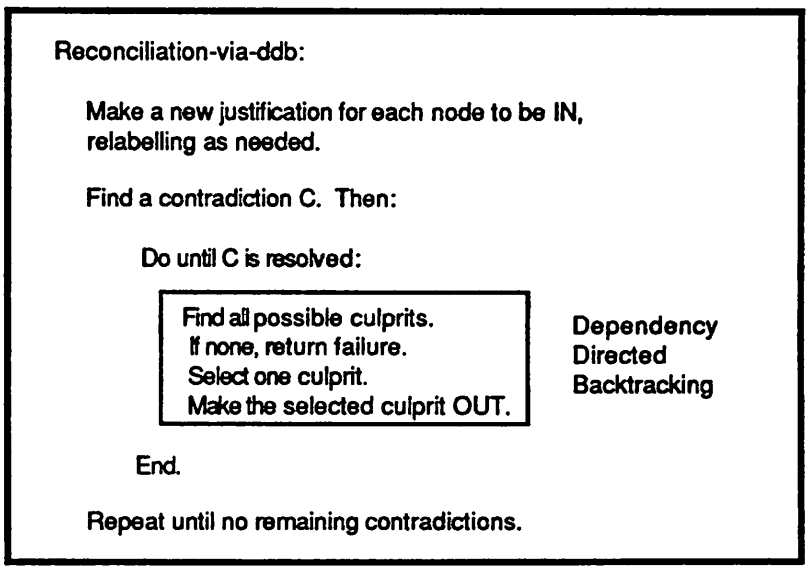

Figure 4.4 Reconciliation via Dependency-directed Backtracking

that actually removes the original contradiction; since this "solution" may have introduced other contradictions into the TMS, it may still be incomplete. Then, further call(s) to DDB must be made (as shown Figure 4.4).

The justifications created during DDB take a carefully crafted form that achieves three goals simultaneously: it allows DDB to be accomplished via nonmonotonic reasoning, it ensures that nodes brought IN by new justifications stay IN only as long as necessary, and it ensures that contradictions stay OUT once they are made OUT. Consider the example in Figure 4.5. A situation with one contradiction is shown to the left. The possible culprits for this contradiction are {A, B}; if B is the selected culprit, then there are two ways of making B OUT: via ¬B or via E. If ¬B is the choice for making B OUT, then ¬B is given

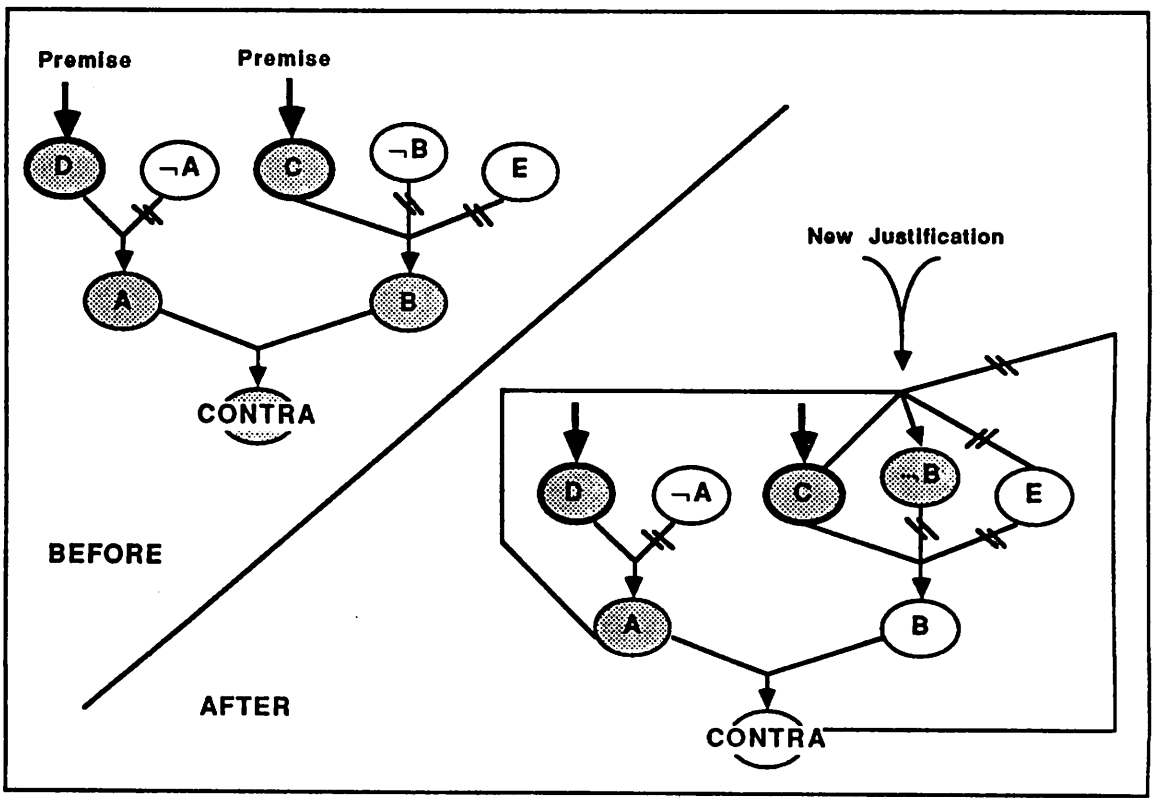

Figure 4.5 Justification Used in Dependency-directed Backtracking

one of these carefully crafted justifications (shown to the right using the method in [Petrie, 1987]). The justification works in three ways as follows.

First, the justification is designed to keep ¬B IN only as long as necessary. For example, if A goes OUT, then the contradiction will go OUT and it is not necessary to keep ¬B IN. If C should go OUT or E should come IN, then B and the contradiction both go OUT, so again it is not necessary to keep ¬B IN. Second, the justification is designed to capture the nonmonotonic reasoning that lead to the original choice of culprit; this means that the right culprits will be generated if it is later discovered that ¬B was the wrong choice. If it is later discovered that ¬B is a possible culprit for some other contradiction, then it can be made OUT through making E IN (using another carefully crafted justification involving A, C, the other new possible culprits, and both contradiction nodes) which will still keep the original contradiction OUT. If it is later still discovered that E leads to yet another contradiction, A (but not E) will be among the possible culprits generated for consideration in dealing with that contradiction. So, even if E comes IN, the contradiction still stays out, now through A being OUT. Thus, the third purpose is achieved: to keep contradictions from recurring.

While DDB is both conceptually elegant and computationally efficient in cases where there are many possible solutions, it has one drawback—there is no way to compare competing solutions against one another. It is only possible to consider individual choices (that will eventually contribute to the solution) in some preferred order. The original implementation actually made choices blindly; extensions to DDB that allow for guidance of local decisions appear in [Petrie, 1987]. Even if there is some knowledge to guide local choices, local decisions may not aggregate into the best global decision

4.2.2 An Alternative Approach to Reconciliation

In this section we describe an alternative approach to reconciliation that attempts to achieve a more complete integration of new information into the TMS. A new approach is made possible by the fact that, in this application, the set of justifications is complete. That is, there are no new justifications (e.g., interrelationships among nodes) being discovered dynamically. This means that all the possible reasons for a node being IN are explicitly available when it is discovered that the node is OUT and should be IN.

4.2.2.1 Making Invalid Justifications Valid

The approach to reconciliation based on relabelling and DDB misses an important opportunity. Each node to be brought IN during reconciliation is simply provided with a new justification *without re-assessing why the node failed to be IN*. That is, no attempt is made to bring the node IN by making one of its currently invalid justifications valid. Finding a rationale for the node being IN using the existing justifications could result in the revision of node labels that would otherwise be unchanged; these changes could be as important as those that are due to the consequences of the node being IN.

An example appears in Figure 4.6. The initial situation is shown to the left; given no observations, only ¬E holds. Two ways of handling a new observation of A are shown. In the middle is the standard method: support A with a new justification and propagate the consequences so that Q now holds, calling DDB if needed (but it is not necessary in this example). To the right is another method: attempt to find a rationale for A, which leads to revising the truth status of both E and C in addition to Q. This is an application of *abduction*, where given a rule P —> Q and the observation of Q, P is inferred. (Note that abduction is not logically sound; given P —> Q, only the deduction of ¬P from ¬Q is sound.) In this case, abduction has been applied twice, once on the observation of A and once on C.

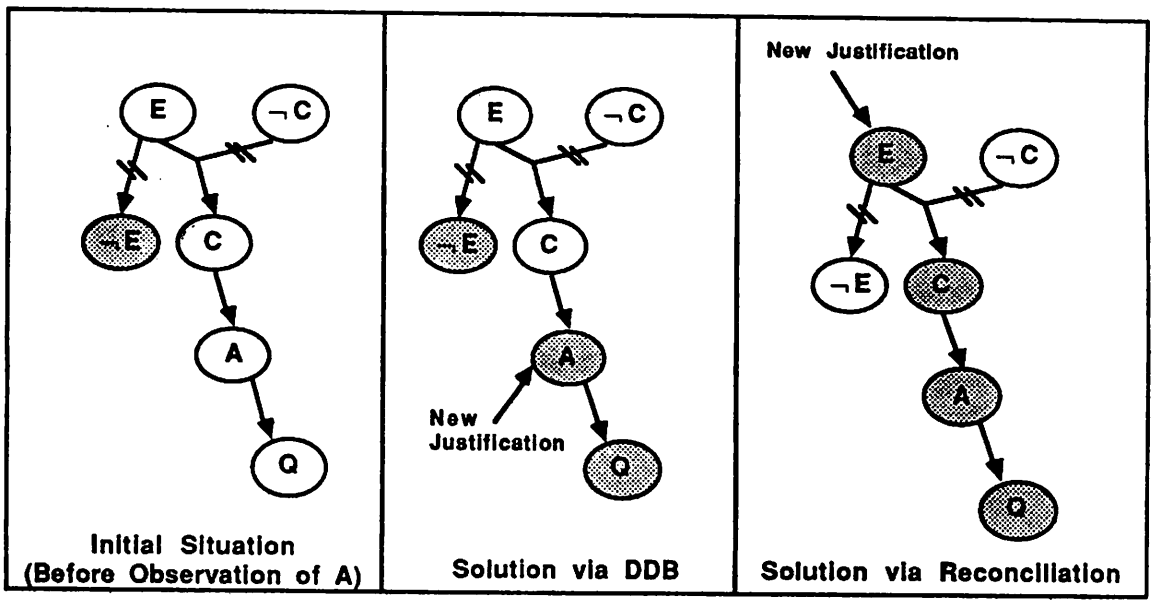

Figure 4.6 Two Approaches to Reconciliation

The method shown in the middle of Figure 4.6 is the only reasonable recourse if the reasons for A being IN are incomplete. In that case, it is not logical to force one of the existing reasons to hold, because it is equally likely that A is IN because of some missing reason. The method on the right is only valid when it is known that the set of reasons (justifications) is complete.

4.2.2.2 Reconciliation as Nonmonotonic Reasoning

This idea of making invalid justifications valid to "explain" new observations is itself based on nonmonotonic reasoning. That is, we are implicitly assuming that if a node is observed to be IN, then *typically* it will be so for one of the known reasons (e.g., one of its existing justifications ought to be valid). The interesting thing is that this type of default has a structure that cannot be expressed in TMS form but can be expressed in the more

general default logic [Reiter, 1980] form. (Thus, default logic may provide the means to formalize the notion of abdunction.)

Consider an example where a node P is the conclusion of two justifications, a monotonic one with support node Q and a nonmonotonic one with support node R and exception node ¬P. In default logic form, these are:

(1) Q—>P and (2) R:MP/P

If we learn that P is IN, then typically either Q should be IN or R should be IN. This is a default of the form:

(3) P: M (Q or R)/(Q or R)

which is not expressible in TMS terms due to the use of "or". Using two separate rules (such as P:MQ/Q and P:MR/R) is not logically equivalent to (3).

The behavior of these rules under default logic correlates well with the concept of reconciliation that we are describing. Notice that if both ¬Q and ¬R hold monotonically (without recourse to default rules), then rule (3) cannot be applied; in this case there is no way to "explain" P (in the TMS, it must simply be supported directly with a dummy justification). If ¬Q holds monotonically, then R is the only viable choice for explaining P, and vice versa if ¬R holds monotonically. In these cases, there is only one possible explanation. However, if there are applicable defaults to support ¬Q, then these defaults compete with rule (3); default logic provides no way to pick an alternative. Our reconciliation algorithm will resolve this choice using domain-independent methods, including giving precedence to rule (3).

This simple example suggests that default logic might be used to formalize the type of reasoning we are suggesting, and once again, underscores the need to study the formal relationships between TMS's and default logic.

4.2.2.3 Extending DDB to Do Reconciliation

The desired behavior in reconciliation can be achieved by extending the ideas of DDB to include examination of currently invalid justifications for ways that they could be made valid. Further, it is easy to "turn off" this extra feature. When a justification is declared, it can be marked as *not-validatable;* then, the system will never attempt to make it valid when it is invalid. If all justifications are *not-validatable*, then the reconciliation algorithm would produce the same results as DDB.

One consequence of the extra processing is that the reconciliation algorithm is considerably more complex than that for DDB. In particular, it must deal with loops in the TMS and must avoid looping forever; DDB avoids loops as a by-product of following only valid justifications.

If reconciliation is to be implemented as an application of nonmonotonic reasoning, then justifications generated during reconciliation must have a carefully crafted form. The form used in DDB applies only to the problem of making valid nonmonotonic justifications invalid. It is not at all obvious what information needs to be represented when invalid justifications are being made valid. The problem is that while only one change is needed to invalidate a justification, generally a set of coordinated changes is required to validate a justification. And, there is another consideration as well. In this application, the TMS is being used to model successive states of a changing world, not successive approximations of the same world state. Unlike contradictions, which are meant to stay OUT forever, the nodes to be brought IN are not meant to stay IN forever; it must be possible to "override" individual justifications generated from earlier reconciliations in later calls to reconciliation. Given these factors, and the inherent complexity of handling both valid and invalid justifications, the algorithm described below is not achieved through nonmonotonic reasoning, although this is an avenue definitely worth exploring.

4.3 An Algorithm for Reconciliation

The key difference between the two approaches to reconciliation has to do with the treatment of invalid justifications. A summary of all differences is as follows:

- In DDB, only those justifications that are currently valid are traversed. In reconciliation, invalid justifications are traversed as well.

- The algorithm for reconciliation given here is not *implemented* via nonmonotonic reasoning as DDB is; it lacks the efficiency of DDB in cases where there are a large number of possible solutions.

- In addition to the DDB method of creating new justifications that are valid, reconciliation also allows for the "blocking" of existing valid justifications to make them invalid; only nonmonotonic justifications are allowed to be blocked. Usually, blocking a justification is done to allow a competing justification to become valid, and therefore corresponds to "changing extensions" in default logic terms (see section 3.1.3.1).

- Reconciliation uses the explicit distinction made between nodes that are certain and those that are by-assumption to narrow the search for possible solutions. Nodes that are certain cannot have their labels changed, so it is useless to examine the inferences on which the status of such nodes is based. Without explicit representation of which nodes are certain, these inferences must be checked. If no nonmonotonic justifications are found, the search has been wasted; even if nonmonotonic justifications are found, they cannot be used to change the status of the node anyway, so again the search is wasted. This distinction is also used to rule out some choices earlier than is otherwise

possible. For example, it is wasted effort to install a new justification on a certain node that is OUT or to attempt to make a justification valid when it has a support node that is OUT and certain. In DDB, these types of choices would be attempted, and then later found to be inappropriate.

- DDB consists of cycles that include analysis, selection among alternatives and installation of the selected alternative; on each cycle, only one change is made to the TMS; cycles are repeated until the first solution is found. While this approach could be used for reconciliation (after adaptation to handle loops), we have experimented with an alternative. The alternative involves identifying as many changes as possible on each cycle; then, domain-independent heuristics are used to choose among sets of coordinated changes, rather than single changes. The cycle may have to be repeated since the changes could be incomplete. Although this approach has the advantage of allowing for the comparison of "nearly complete" solutions, its performance is not acceptable if large numbers of solutions exist.

- Reconciliation is designed to work in the presence of rules in normal and seminormal form, even loops, and preference rules. It also handles "otherwise" rules.

When the reconciliation algorithm is called, its input is a list of nodes that must be made IN; all these nodes must be by-assumption rather than certain. The list is determined by the plan recognizer, as described in section 4.1.4.1. In general, the list will include some nodes that are already IN; it is a constraint on reconciliation that these nodes be kept IN by whatever reconciliation solution is finally selected. For each of the nodes on the list, the node representing its negation may be IN (in which case the proposition represented by

the node is now false) or OUT (in which case the proposition is now unknown). Obviously, reconciliation must avoid creating contradictions in the TMS; this means that whenever a given node is explicitly made IN, its negation must also be made OUT.

Reconciliation includes both identifying and installing a solution. It is possible for reconciliation to fail. An example is shown in Figure 4.7. There is no way to bring both N1 and N2 IN simultaneously, given that A and ¬B are both IN with-certainty. A failure corresponds to an attempt to adopt assumption(s) that are inconsistent in the current world state. When reconciliation fails, the TMS left in the state it was in when reconciliation was called, and notice of the failure is returned to the plan recognizer.

Reconciliation consists of two parts: analysis of candidate solutions, and selection of a candidate. These are described below.

4.3.1 Analysis of Candidate Solutions

Reconciliation begins with the identification of all candidate solutions. This identification is made easier by the fact that, in the present implementation, the TMS keeps *lists* of all valid justifications supporting IN nodes, not just a single justification. This

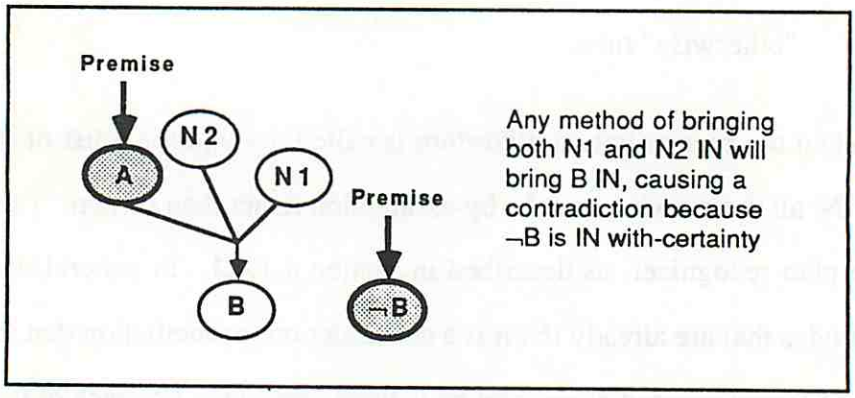

Figure 4.7 Reconciliation Fails

means that it is possible to identify in advance all changes that are needed to achieve a given solution (unless the solution involves interacting changes or contradictions, as discussed in section 4.3.2.3). When only one valid justification is recorded for each IN node, identification of all changes in a given solution only becomes apparent in stages, with each stage being installed (thus changing the TMS labels) before the next stage can be identified.

During identification of candidate solutions, the TMS is not changed in any way. The fact that there can be interactions within a solution is ignored at this time, but is addressed later in the reconciliation process (see section 4.3.2.3).

4.3.1.1 Bringing Nodes IN or OUT

In order to bring a node IN, it is necessary to make *one* of its justifications valid. This means finding a justification in which the only support nodes that are OUT are OUT by-assumption, and the only exception nodes that are IN are IN by-assumption (we say that a justification is *activatable* if it meets this criterion). To make an activatable justification valid, *all* of the OUT support nodes must be made IN *and all* of the IN exception nodes must be made OUT. Justifications that are declared to be *not-validatable* cannot be made valid, and so they are never activatable.

In some cases, a node may have more than one activatable justification. Only one of these actually needs to be made valid to bring the node IN. Therefore, each activatable justification should be treated as a separate alternative.

There is an additional consideration for bringing a node IN—the node representing its negation must be made OUT, if not already so, to avoid introducing a contradiction. If the justification by which a node is being made IN is nonmonotonic, then the negation will be one of the exception nodes (due to the use of normal and seminormal form) and so it will be made OUT in the normal course of making that justification valid. Otherwise, it is

necessary to take additional steps to ensure that the negation goes OUT. (Since the changes needed to make negation go OUT are common to every justification, it is more efficient to do this analysis once, and reuse the information.)

In order to bring a node OUT, one must make *all* of its valid justifications invalid. This means, for each currently valid justification, making *one* support node (that is now IN by assumption) OUT *or* making *one* exception node (that is now OUT by assumption) IN. Only one of these changes will render the justification invalid, so each change is represented as a separate alternative. However, one such alternative is necessary for all valid justifications, because unless all are made invalid, the node won't go OUT.

Bringing a node OUT is not entirely symmetric with bringing a node IN: while bringing a node IN requires that its negation *must* also go OUT (in order to avoid a contradiction), bringing a node OUT does not *require* a change in the status of the negation. Of course, when making a nonmonotonic justification invalid, the negation of the conclusion node will be one of the exception nodes, so making the negation IN will be one of the alternatives under consideration (the dual cannot be IN with-certainty because that would imply that the node itself is certain, and reconciliation can only take place on a list of by-assumption nodes).

4.3.1.2 Primary Nodes and Justifications

Analyzing a node for the ways of bringing it IN or OUT produces a list of other nodes that need changing. The ways of bringing these nodes IN or OUT must then be analyzed in turn, and so on recursively. (This process is actually carried out depth-first, not breadth-first.) In this way, all the different possible solutions will be identified (some of these solutions may be incomplete or not viable, as discussed in sections 4.3.2.3).

Since there are loops in the TMS, the recursive process of finding the primary nodes and justifications must be prevented from recursing infinitely. (DDB conveniently avoids this problem because it only considers valid justifications.) This is done by keeping track of which nodes have already been visited. Whenever a new step on the path reaches a node which has already been visited, analysis stops. The treatment of such a node is discussed further below.

Analysis also halts in two other ways: when it reaches a node that cannot be made IN through changes to other nodes, or when it reaches a justification that cannot be made invalid through changes to other nodes (how each of these things happens is described in section 4.3.1.4 below). These are called primary nodes or primary justifications. Just because a node has some primary justifications does not means that all its justifications are primary; some may be invalidatable through changes to other nodes. Therefore, it would be misleading to use the designation "primary" on nodes going OUT.

4.3.1.3 Changing TMS Justifications

Each primary node and each primary justification (in the selected solution) will lead to a change in the TMS justification structure; these changes will then cause the desired changes to the IN/OUT status of nodes in the TMS. Before we can determine the form of these changes for this application, it is necessary to consider their impact on future states of the world. If we didn't care about future states of the world, it would be perfectly satisfactory to support a primary node monotonically and simply delete a primary justification. Recall that the pre-declared justifications are written to embody the proper time-varying characteristics as described in section 3.3.1.6. Now we have to consider what time-varying characteristics to put into supporting a primary node or blocking a primary justification.

Our choices are based on four assumptions about persistence of changes. First, that a node X explicitly brought IN in state N (via a dummy justification) persists in the states that follow unless new information arriving in a later state makes a competing justification valid (if the competing justification is nonmonotonic, this can only happen if there is an applicable preference for ¬X). Second, a block explicitly made to a justification in state N persists, but does not prevent new information arriving in later states from making another justification for the same conclusion valid. Third, the changes made to adopt an assumption explicitly or to block a justification explicitly can only be "undone" by a call to reconciliation in a state subsequent to N. And fourth, changes can be "undone" individually Informally, dummy justifications and blocks in justifications stay until removed by a subsequent call to reconciliation, and they need not be removed as a group. However, nodes that are explicitly forced IN or OUT may go OUT or IN in later states, even while the dummy justifications and blocks in justifications are in place.

These persistence assumptions constrain the form to be used to bring primary nodes *IN*—they must go in nonmonotonically using normal/seminormal form. Note that a nonmonotonic justification makes its conclusion IN by-assumption, not IN with certainty; this means that in later snapshots of the world, it will still be possible to revise the label on this node during reconciliation. Note also that unless a nonmonotonic justification is in normal or seminormal form, a contradiction will result if the negation of its conclusion comes IN monotonically. This method of bringing a primary node IN is shown in Figure 4.8. Since the new exception node is OUT, the justification automatically become valid after the negation of the conclusion is made OUT, or immediately if the negation of the conclusion is already OUT.

The persistence assumptions are met if a primary justification is "blocked" in the obvious way—by adding a new support node in the justification as shown in Figure 4.9. Since the new node has no justifications, it is OUT, which makes the justification invalid.

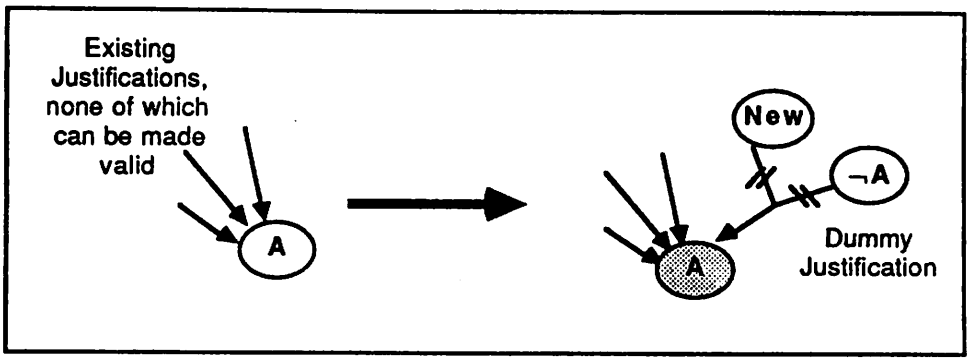

Figure 4.8 Form of Dummy Justification

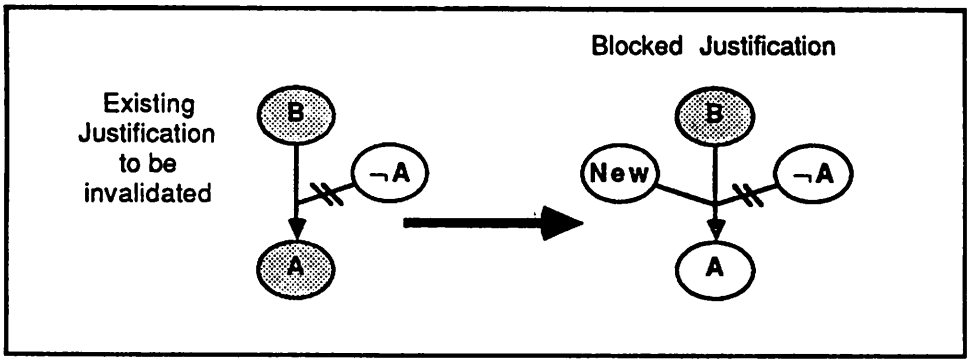

Figure 4.9 Form of Blocked Justification

This change is actually accomplished by deleting the existing justification and immediately re-creating another justification with this new support node in addition to all the original nodes.

Unique names are automatically generated for the new nodes created in the dummy and blocked justifications. Currently, the node name includes the sequence number of the current action (which is a kind of timestamp). Later, one can recover the fact that this change in the TMS was made to accommodate that action. Also, this information could be

used to help select among alternative solutions—for example, by biasing selection for/against undoing the most recent changes.

As mentioned previously, the nature of these changes to make primary nodes go IN and to invalidate primary justifications is such that the algorithm for reconciliation is not an application of nonmonotonic reasoning.

4.3.1.4 Finding Primary Nodes and Justifications

In the absence of loops, primary nodes that need to be forced IN occur in two ways. First, a node that is not the conclusion of any justification cannot be made IN via other nodes—there are no other nodes to change. Second, it may be impossible to make any existing justification for the node valid because all such justifications are not activatable—either they are not-validatable or they have at least one support node that is OUT with certainty or one exception node that is IN with certainty.

The only way to have a primary justification is due to a loop. If a node is to be forced OUT (and the node itself has not been encountered before), there is always a way to make it OUT through other nodes unless those nodes have been encountered before. The following argument shows why. Let N be the node to be forced OUT. N cannot be supported by a monotonic justification involving only certain nodes, or it would be certain itself; therefore, any currently valid monotonic justification for N has at least one by-assumption node (among its support nodes) that can be changed. If N is supported by a nonmonotonic justification, which must be in normal or seminormal form, the negation of N is guaranteed to be one of the exceptions. Further, the negation must be by-assumption because N is. That means that, again, there is at least one by-assumption node that can be changed. Thus in every currently valid justification for N, whether monotonic or nonmonotonic, there is some other node that, if brought IN, will force N OUT. So unless we have encountered these nodes before, N can be forced OUT by changes to them.

When the analysis encounters a node N in an even loop, it will traverse the loop until it returns to N, the node at which it entered the loop. What happens then depends on the type of change that was being attempted when N is re-encountered. There are 4 cases, shown in Figure 4.10. These are described as follows:

- Case 1: Attempting to bring node M IN via justification J in which N appears as a support node.

  To make M IN via J, N must go IN. Since it cannot due to a loop, J is eliminated as a possibility for bringing M IN. Then, M may become a primary node, if none of its other justifications can be made valid.

- Case 2: Attempting to bring node M IN via justification J in which N appears as an exception node.

  To make M IN via J, N must go IN. Since it cannot due to a loop, J is eliminated as a possibility for bringing M IN. Then, M may become a primary node, if none of its other justifications can be made

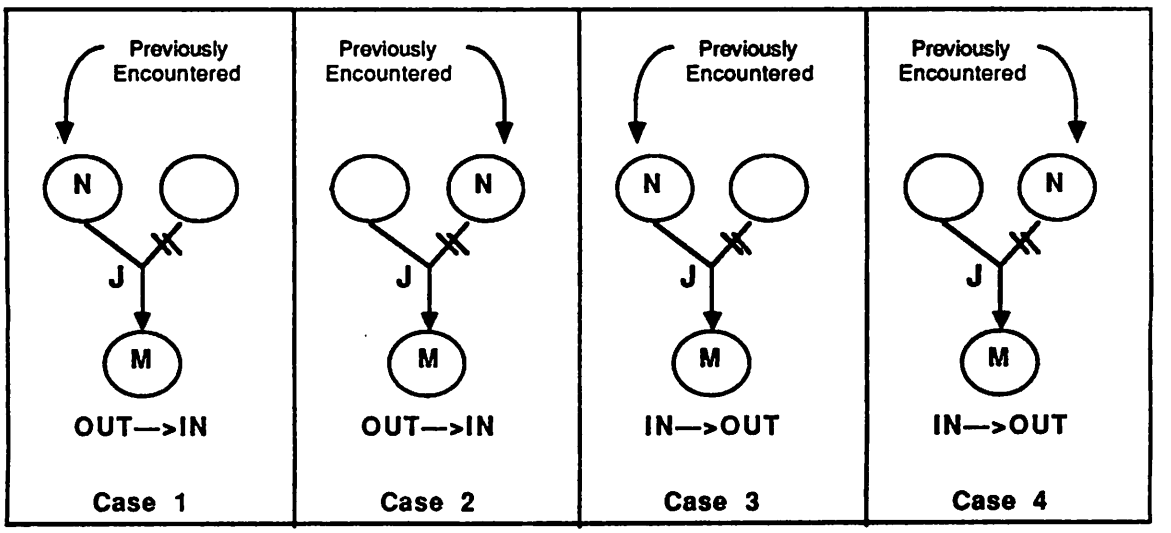

Figure 4.10 Analyzing a Previously Encountered Node

valid. The special considerations that apply when N=¬M are discussed in section 4.3.1.5.3, below.

- Case 3: Attempting to bring node M OUT via justification J in which N appears as a support node.

  To make M OUT, J must be invalidated. Since N appears in a loop, N is eliminated as a possibility for invalidating J. If there are no other ways to make J invalid (i.e., if all its other nodes are certain or in loops), then J becomes primary.

- Case 4: Attempting to bring node M OUT via justification J in which N appears as an exception node.

  To make M OUT, J must be invalidated. Since N appears in a loop, N is eliminated as a possibility for invalidating J. If there are no other ways to make J invalid (i.e., if all its other nodes are certain or in loops), then J becomes primary. The special considerations that apply when N=¬M are discussed in section 4.3.1.5.3, below.

4.3.1.5 Special Cases

Five special cases need to be discussed. They relate to deleting dummy justifications, unblocking blocked justifications, loops from normal/seminormal form, interactions with preferences, and handling "otherwise" rules.

4.3.1.5.1 Deleting Dummy Justifications

A special case arises when a node to be made OUT is supported by a dummy justification created during a previous invocation of reconciliation. Normally, the analysis will proceed to notice that the way to invalidate the dummy justification would be through its exception nodes. A more desirable solution is simply to delete the dummy justification which was originally created but is now no longer needed. Thus, when a dummy justification is encountered during analysis to make a node OUT, a note is made that it needs to be deleted.

4.3.1.5.2 Removing Blocks in Justifications

Another special case arises when a node to be made IN is the conclusion of a justification that was blocked during a previous invocation of reconciliation. The analysis will proceed to notice that one thing required to make this justification valid is to support the blocking node in the justification. This will require adding a dummy justification for the blocking node. A more desirable solution is simply to modify the justification to remove the blocking node. Thus, when a blocked justification is encountered during analysis, and it is found to be activatable, a note is made that it needs to be unblocked in addition to whatever else is required to make it valid.

4.3.1.5.3 Loops in Normal/Seminormal Form

The use of normal/seminormal forms leads to the frequent occurrence of loops with a particular topology. Some special considerations affect how these loops are handled. The prototypical cases of loops from normal/seminormal justifications are diagrammed in Figure 4.11; in each case, N is the first node to be encountered in the loop. These cases are described as follows:

- Case 1: Attempting to force node N IN via justification J1, which requires that node ¬N go OUT (in fact, no matter how N is brought IN, ¬N will have to go OUT to avoid a contradiction). No special considerations apply here. All justifications concluding ¬N will have to be invalidated, including J2. Due to the loop, J2 cannot be invalidated through N. Therefore, either J2 can be invalidated through its other nodes (here, just B), or J2 will have to be blocked. (This situation of J2 corresponds to Case 4 above, where M=¬N.)

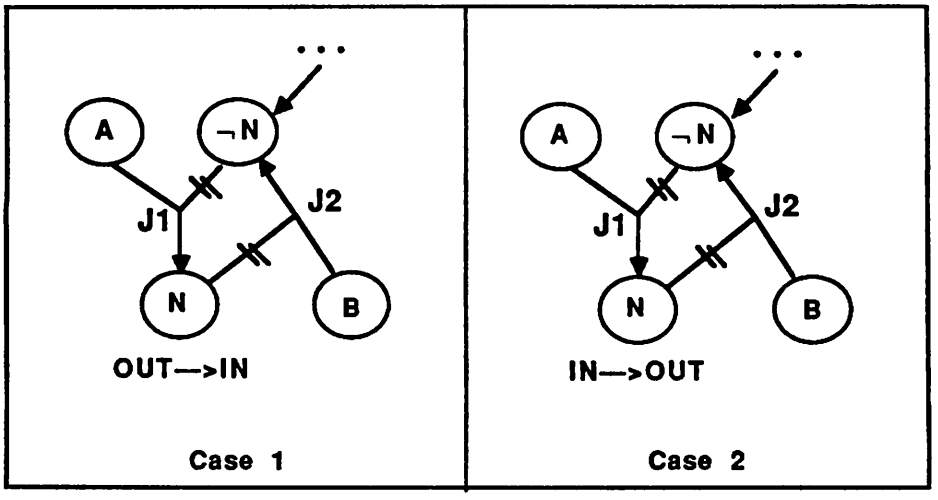

Figure 4.11 Prototypical Loops

- Case 2: Attempting to force node N OUT, which requires invalidating justification J1, one option for which involves bringing ¬N IN. (This situation of J2 corresponds to Case 2 above, where M=¬N.) There are special considerations to handling this option. And it is necessary to divide this case into four subcases as follows:

  Case 2A: Suppose there is a way to bring ¬N IN monotonically, (a monotonic justification for ¬N will not have N in its support nodes); then, the desired effect is achieved directly—¬N goes IN without consideration of N and once ¬N is IN, J1 is invalidated.

  Case 2B: Suppose that justification J2 already has all its support nodes IN and all its exception nodes OUT, except that exception node N is IN. In this case, blocking justification J1 will achieve the desired effect, by making N OUT which allows J2 to become valid.

  Case 2C: Suppose that there is a way to bring all the support nodes of J2 IN and all the exception nodes OUT, while excluding consideration of N. If this can be done, then the situation reduces to that of Case 2B. Therefore, we add the blocking of J1 to the partial solution for the nodes of J2 to arrive at a complete solution.

Case 2D: Suppose there is no way to bring all the support nodes of J2 IN and all the exception nodes OUT (excluding consideration of N). Then, there is no way to force ¬N IN via J2.

4.3.1.5.4 Interactions with Preferences

There are two relevant questions about interactions with preferences: what is the effect on the preferences of installing a solution, and does the existence of preferences affect the generation of solutions (e.g., could a preference simplify or destroy the solutions which are being generated?). We consider these in turn.

The first question is simple: installing a solution can make previously inapplicable preferences become applicable. An example of how this happens is shown in Figure 4.12. Initially, both P and Q are OUT; the otherwise justification brings ¬N IN. Later, a change made elsewhere in the TMS brings Q IN, which bring P IN; but, justification J is invalid because ¬N is IN, which runs counter to the preference for N. Therefore, it is necessary

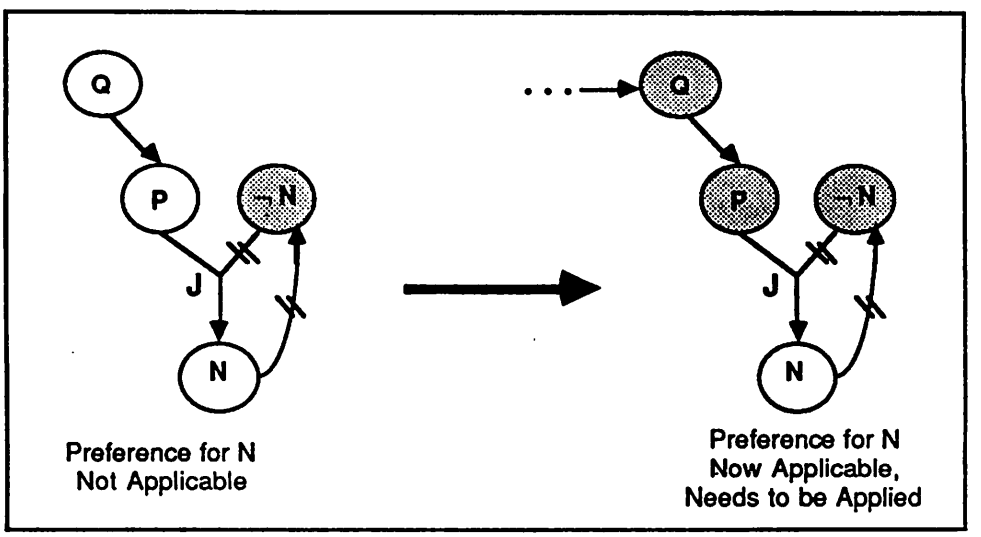

Figure 4.12 Reconciliation Requires Reinstallation of Preferences

to re-establish preferences after a solution is installed, but before the solution is checked for success.

To explore the question of the effect of preferences on solutions analysis, we must look again at examples of even loops, the configurations that give rise to the multiple labellings that are the subject of the preferences. There are two cases (see Figure 4.13), each showing the simplest kind of even loop which currently supports two labellings. (The multiple labellings may have existed before reconciliation was called, or the reconciliation analysis may be creating them.)

Case 1: Attempting to bring N IN, where N is in an even loop currently supporting two labellings. There may be ways of doing so via justifications other than J1 (e.g., monotonic justifications), but the J1 case is the case where the preferences arise. To make N IN via J1, A must come IN (if not already IN). Then, the multiple labellings issue can

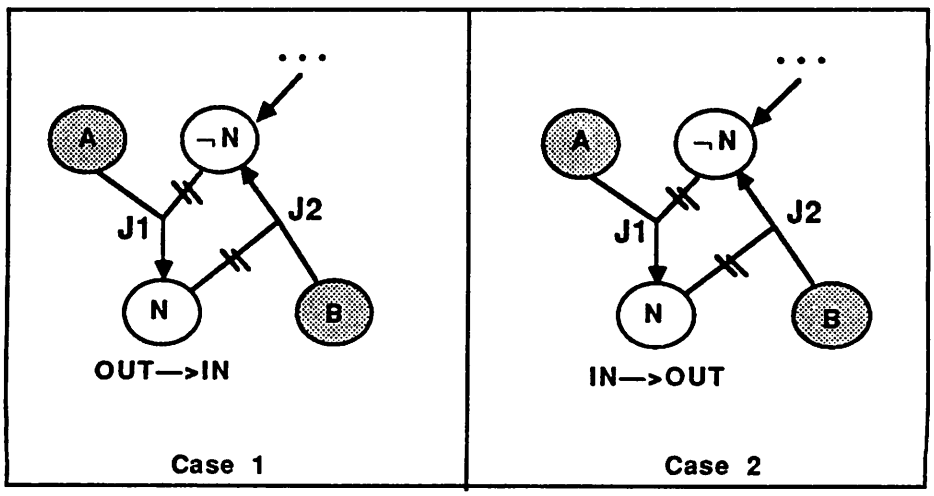

Figure 4.13 Analysis under Preferences

arise, and the preferences, in some cases, reduce the number of changes needed to actually bring N IN. There are five subcases:

Case 1A: There is an unconditional preference for N. The desired labels are consistent with this preference. Then, if ¬N is supported only by nonmonotonic justifications, nothing more needs to be done; these justifications (such as J2) will be invalid due to this preference. However, if ¬N is supported by any monotonic justifications, these must be made invalid.

Case 1B: There is a conditional preference for N. The desired labels are consistent with this preference, but we can't tell if the conditional will hold after the solution (which is only partly identified at this point) is installed. Therefore, we must be conservative, and explore all the ways to make ¬N go OUT, which involves invalidating all justifications for ¬N.

Case 1C: There is an unconditional preference for ¬N. The desired labels run counter to this preference. Therefore, we must explore all the ways to make ¬N go OUT, which involves invalidating all justifications for ¬N.

Case 1D: There is a conditional preference for ¬N. The desired labels run counter to this preference, but we cannot yet tell if the conditional will be true after this solution (which is only partially identified at this point) is installed. Therefore, we must be conservative and explore all the ways to make ¬N go OUT, which involves invalidating all justifications for ¬N.

Case 1E: There are no preferences on this loop. Then, the current labels were chosen arbitrarily. In order to ensure that the "right" labels are established, we must force ¬N OUT, which involves invalidating all justifications for ¬N.

Case 2: Attempting to bring N OUT, where N is in an even loop currently supporting two labellings. There may be ways to invalidate J1 through nodes other than ¬N; if so, the loop will no longer support two labellings, and there is no interaction with the preferences in that case. When attempting to invalidate J1 through ¬N, A will be staying IN and B will be coming IN (if not already IN). Here there is a potential for

preferences to work against solutions, and so extra changes (blocks in justifications) may be needed. There are five subcases:

Case 2A: There are no preferences on this loop. Then, the current labels were chosen arbitrarily. In order to ensure that the "right" labels are established, there must be a block placed on J1.

Case 2B: There is an unconditional preference for N. Then, the desired labels (N OUT) will violate this preference. In this case, it is necessary to block J1.

Case 2B: There is a conditional preference for N. Then, the desired labels (N OUT) may violate this preference. Since we can't yet tell if the conditional on the preference will be operative or not after the solution is installed (the solution is incompletely identified at this point), we must be conservative, assume it will be operative, and therefore install a block on J1. This block may actually turn out to be unnecessary.

Case 2B: There is an unconditional preference for ¬N. (This case can only arise if the reconciliation algorithm is making J1 valid.) Here, the preference is for the desired labels (¬N IN). Hence, it is unnecessary to block J1.

Case 2C: There is a conditional preference for ¬N. (If there is such a preference and N is now IN, then either N is supported currently by a monotonic rule or the preference is conditional and the condition does not hold.) Since we can't yet tell if the conditional on the preference will be operative or not after the solution is installed (the solution is incompletely identified at this point), we must be conservative, assume it will not be operative, and therefore install a block on J1. This block may actually turn out to be unnecessary.

4.3.1.5.5 "Otherwise" Rules

Two special considerations apply to the handling of "otherwise" rules.

First, the reconciliation algorithm will not create a dummy justification for a node that is the conclusion of an "otherwise" rule (EXCEPT ¬N —> N). Making N IN in this case requires making ¬N OUT. If ¬N has already been visited on the path, then the standard approach to breaking the circularity would be to create a dummy justification to support N.

This normally corresponds to the explanation that N is IN for reasons not covered in its other justifications. But, a justification in otherwise form is actually more general than a dummy justificaiton. Therefore, a more attractive approach is to block each justification that is keeping ¬N IN; this will allow the "otherwise" rule to bring N IN. This corresponds to the explanation that every currently valid justification for ¬N has an additional condition of applicability previously unidentified.

Secondly, the reconciliation algorithm will never block an "otherwise" justification supporting a node N. When analysing the way to bring N OUT, it will be noticed that the only way of invalidating an otherwise rule is through the exception node; this will always involve bringing ¬N IN. Then, if ¬N is coming IN by making a monotonic justification valid, then that justification will "override" the "otherwise" rule, and the desired effect will be achieved without a block in the otherwise justification. On the other hand, when ¬N is coming IN by making a nonmonotonic justification valid, the "otherwise" justification will prevail unless something else is done. The obvious solution is to block the otherwise justification. But, preferences have to be re-established anyway after installing a solutions (as described above), and PREFER ¬N is guaranteed to be among the preference rules. Therefore, the re-establishment of preferences will take care of the final step in invalidating the otherwise justification, and it will be unnecessary to block it explicitly.

4.3.1.6 Representing Solutions

As the foregoing analysis shows, a potential solution to reconciliation cannot simply be represented by two lists of nodes: those to go IN and those to go OUT. A complete set of changes must actually be represented by three lists: one for nodes to go IN (accomplished by generating dummy justifications to these nodes), one for justifications to be blocked, and one for dummy justifications to be deleted and blocked justifications to be unblocked. These lists are called the IN, BLOCK, and UNDO lists respectively. Primary

nodes make up the IN list, and primary justifications make up the BLOCK list. (Actually a fourth list, the REVERSE list, is used internally in the algorithm in Appendix F, but it is not regarded part of the final answer.)

Each change called for by each entry on all three lists is required to accomplish a single solution to reconciliation. Generally, there will be multiple alternative sets of such changes. Thus, the final representation of choices is a list of alternatives, where each alternative consists of a list of three lists (the INs, BLOCKs, and UNDO's).

4.3.2 Choosing a Solution

In this section we consider how to choose the best candidate solution. Since the solutions were arrived at without consideration for interactions, once the candidate is installed, it must be checked to see that it is in fact a complete and viable solution. If so, it is accepted; otherwise, there is more work to be done.

4.3.2.1 Selecting a Candidate Solution

There are several possible heuristics for selecting the best candidate solution. The state of the TMS prior to reconciliation represents the most informed view of the current state of the world. Therefore, it seems only rational to choose a revised view of the world which is closest to the most informed view. However, "closeness" between two views can be measured in several different ways. One way is simply to count the number of nodes whose labels differ in the two views. Another way is to count the difference in the number of dummy justifications and blocked justifications in the two views (where each dummy or blocked justification represents a "patch" accounting for an exception in the domain knowledge captured in the justifications). These two ways of measuring closeness are different—adding a single dummy justification can have greater or lesser impact on node labels depending on the connectedness of the node being supported.

The selection criterion which we describe here is the minimal patches criterion. The primary motivation for using this criterion is that it "does the right thing" in cases where multiple nodes are being brought IN and there is a common explanation. For example, a common explanation might involve a single blocked justification, whereas two separate explanations each require a separate dummy justification (as has already been seen in the example in Figure 3.10). A practical motivation in choosing this heuristic is that it can be estimated directly on the IN/BLOCK/UNDO representation of alternatives. Using the minimal label changes criterion requires a lot more analysis.

Each alternative generated during analysis of possible changes is rated as to the number of patches it requires in the TMS justifications. The patch count is equal to the number of unique entries on the IN list plus the number of unique entries on the BLOCK list minus the number of unique entries on the UNDO list. (Duplicate entries arise from commonalities in solutions for two or more nodes.)

4.3.2.2 Installing the Candidate

The changes to be installed are represented as three lists described above. Duplicates have already been removed from the lists, so it is not necessary to check if the specific operation has already been performed. The operations are performed even if the desired effect of the operation (a node going IN or OUT) has already been achieved—because it could be de-achieved by later operations. Installation is completed by re-establishing the preferences (the reasons for this are described in section 4.3.1.5.4 above).

4.3.2.3 Verifying the Candidate

Candidate solutions have to be verified before being accepted. There are three different ways that a candidate could fail: it could fail to bring all the desired nodes IN, it

could introduce contradictions into the TMS, or both. Candidates that fail the verification test are either invalid (these must be rejected) or incomplete (these require further analysis).

Figure 4.14 shows two different types of contradictions that can arise. At the top is a fatal contradiction: as a by-product of changing other nodes (here, changing B in order to change N), a node (here, C) comes IN when its negation is already IN with certainty. A

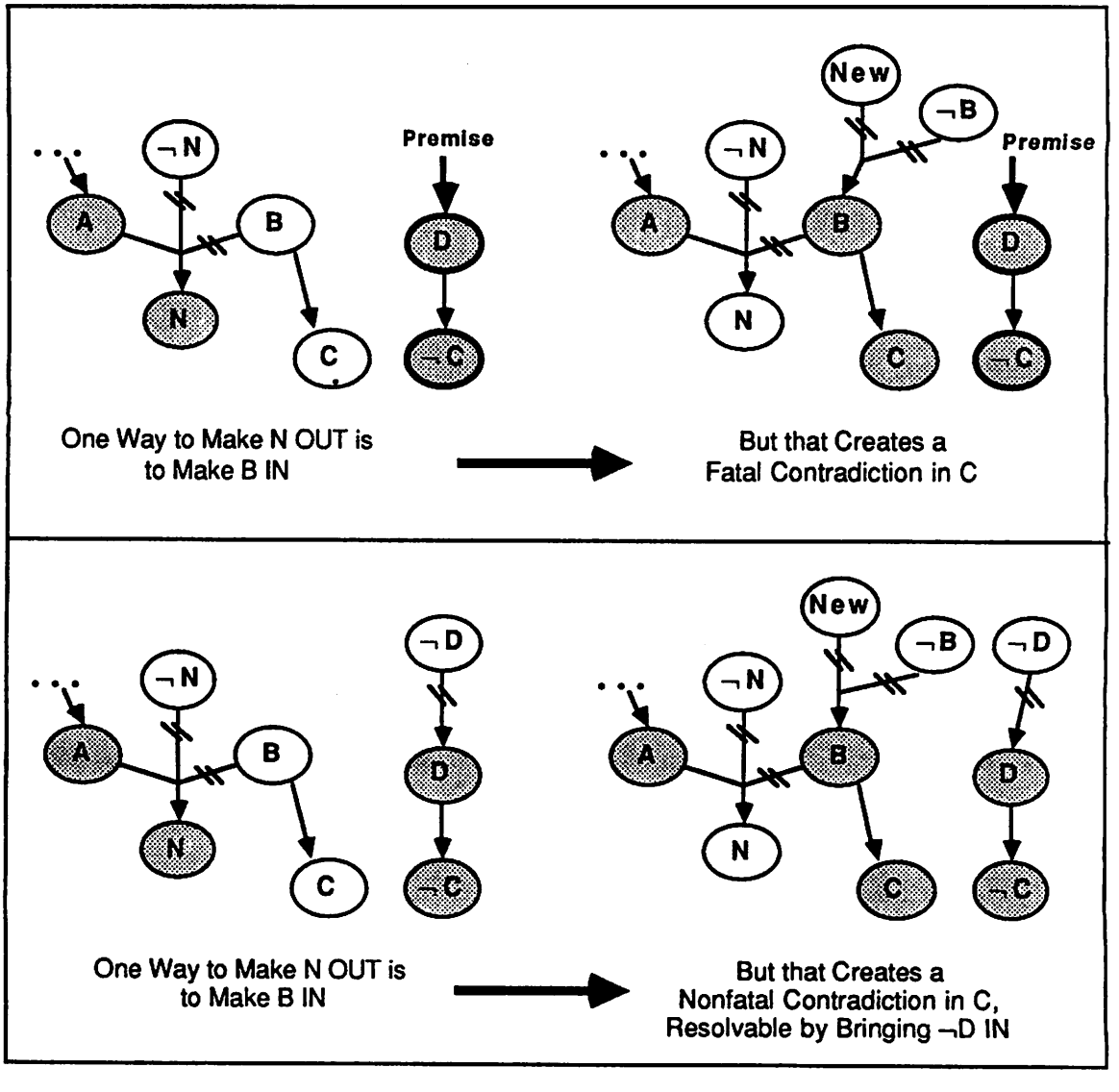

Figure 4.14 Fatal and Nonfatal Contradictions

solution is invalid if it leads to a contradiction where one of the contradictory nodes is IN with certainty. The case at the bottom is an nonfatal contradiction: as a by-product of changing other nodes, a node comes IN when its negation is already IN by-assumption. Such a contradiction can be resolved by making additional changes (here, making ¬D IN). In this situation, we have an incomplete solution that needs to be extended.

Incomplete solutions also arise when interactions among the separate parts of a solution cause one or more nodes that were to be brought IN to remain OUT. An example where there is a negative interaction within a solution is shown in Figure 4.15; there the solution shown will not achieve the desired effect—the choice of method to bring one node IN contributes to bringing another node OUT that was also meant to come IN.

Given this analysis, after a solution is selected and installed, two checks must be made. First, all nodes that were to go IN should be checked to see if they are IN. (In the process of doing this, the number of nodes that are OUT can be counted and used as a

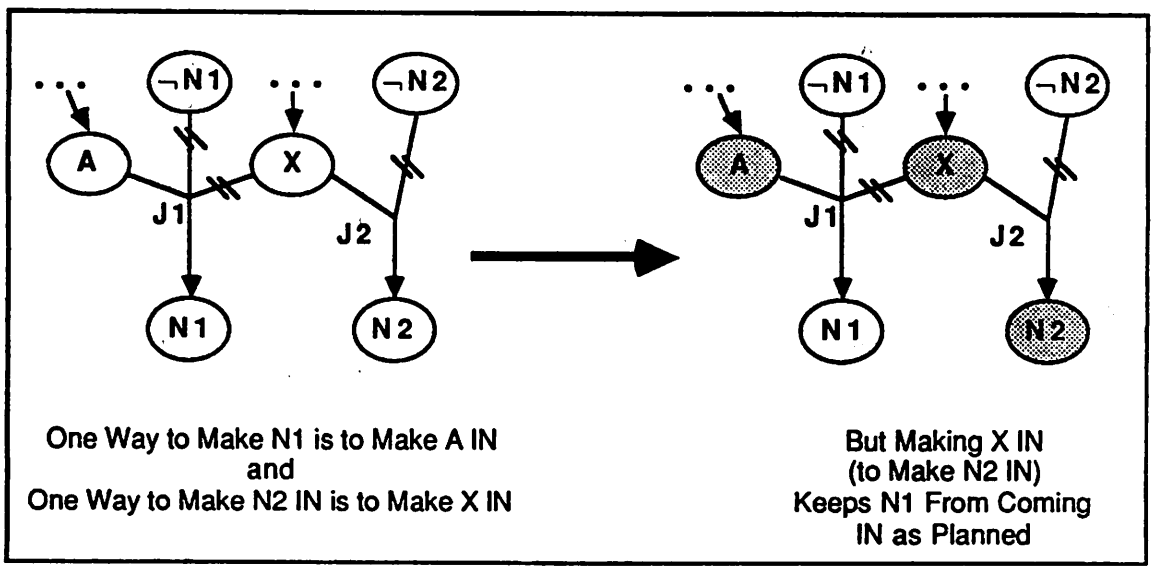

Figure 4.15 An Incomplete Solution

measure of how "incomplete" the current solution is.) Second, all contradictions should be identified and classed as fatal or nonfatal. (In the process of doing this, the number of nonfatal contradictions can be counted and used as a measure of how "contradictory" the current solution is.) A solution is invalid if it has one or more fatal contradictions; all other solutions that fail these tests are incomplete.

Further analysis is required on an incomplete solution—another call to reconciliation with the original list of nodes to go IN, and all the contradictions on a list of nodes to go OUT. (The ability to make a node OUT is already included within the reconciliation algorithm). The changes needed to complete a solution must be made without changing any dummy justifications or blocks in justifications that were part of the original solution (this can be done using the action number label attached to nodes as described in section 4.3.1.3).

4.3.2.4 When Candidates Are Incomplete

The current GRAPPLE implementation will prefer any solution (at whatever patch count) to a solution that is incomplete. Thus, it will successively try all candidates in order by patch count, and accept the first successful one. The unsuccessful ones that happened to bring all the desired nodes IN but introduced contradictions are reserved for further consideration; others are ignored. If all the initial candidates are invalid, then reconciliation fails. If all original candidates are incomplete and there are no candidates that managed to bring all the desired nodes IN, then a default solution involving a dummy justification for each node to be brought IN is installed. If it proves invalid, reconciliation fails; if it is incomplete (e.g. introduces nonfatal contradictions), it becomes the single reserved candidate for the next round of analysis.

If all solutions are incomplete, then the reserved candidates are reconsidered, with those involving the fewest contradictions considered first. If such a candidate can be

completed through one additional pass of analysis, then that candidate is accepted. If all these candidates are found to be incomplete or invalid after one further pass of analysis, then reconciliation fails. (This means that it is possible for reconciliation to fail when a solution does exist, but this only happens for cases where the interrelationships among the nodes are particularly convoluted. Continuation of the search to find solutions when at least one exists is a straightforward extension.)

Because reconciliation is not implemented as nonmonotonic reasoning, either the changes due to a failed candidate must be deleted from the TMS or the state of the TMS must be saved before a candidate is installed and restored afterward if the candidate fails. It is possible to do more analysis of candidates before they are installed to detect in advance that the candidate will fail, or to use information about how a particular candidate failed to identify other candidates that will fail in the same way. However, the analysis is not free, so the cost of doing the analysis must be measured against the saving due to avoiding candidate failures. This is ån area for further consideration.

4.3 Summary

In this chapter, we have supplied the implementation details for our approach to deeper domain modeling. In section 4.1, the technical and algorithmic details were given for marrying nonmonotonic reasoning (about the state of the world) with a plan recognizer. This included the TMS implementation, changes and additions to the basic plan recognition algorithm, the calculation and use of credibility, and the interface to reconciliation. In sections 4.2 and 4.3 we discussed reconciliation, a new approach to revising assumptions in a TMS. The motivation for a new approach was given, the approach was described, and then all the technical issues were enumerated and resolved.

This chapter completes the description of deeper domain modeling. In the next chapter, another type of representational limit in planning formalisms is identified and a method for capturing metalevel domain knowledge is described.

CHAPTER 5
METALEVEL DOMAIN KNOWLEDGE

In this chapter, we present a general solution to a class of representation problems in planning formalisms. In the first section, we give a number of different examples of domain knowledge that cannot readily be expressed in standard operator format. While each of these cases could be accommodated by designing additional features in the operator definition language, as was done in Chapter 2, a more general solution is suggested by considering what is common among them. That common factor is that the knowledge can be compactly expressed as modifications to plans, or, more formally, as metalevel transformations. In the second section, we investigate transformations on plans in more detail. Additional examples are given to show the broad applicability of the transformational approach. In the third section, we show how these transformations can be formalized in metaoperators and metaplans. Finally, we end by discussing the issues of implementing the metaplans in a plan recognizer.

5.1 Limits on Representational Power

Limits on representational power arise in two different ways. The first, and obvious, limitation involves knowledge that simply cannot be expressed. The second, more subtle,

limitation involves knowledge that can be expressed, but only after sacrificing some of the advantages normally associated with operator definitions. Before presenting examples of representational limits, it is helpful to highlight some of the desirable characteristics of operator definitions and libraries that ought to be preserved.

Hierarchical plan systems, based on NOAH [Sacerdoti, 1977] and NONLIN [Tate, 1976], are particularly appropriate for handling the operator libraries of complex domains. The use of hierarchical (i.e., complex) operators allows activities to be defined at multiple levels of abstraction, with more or less detail as appropriate; this provides an orderly approach to covering all domain activities. The hierarchical organization of operators contributes to two valuable features: modularity and independence.

The modularity of operator libraries facilitates some practical aspects of working with large numbers of operators. Following the principles of information-hiding [Parnas, 1972], certain details can be encapsulated in one or a few operators; later, if those details must change, the effects are local, not global. Locality of change is particularly important for operator libraries that are large and have to maintained over time; making changes, such as refining existing operators or adding additional operators with new knowledge about actions, is a manageable activity if changes are localized.

In general, operators can be written without knowledge of the other operators—either those operators that achieve the same or similar goals, or those operators that mention those (sub)goals in their decomposition. When a new operator, satisfying a subgoal of other operators, is added to the library, the existing operators do not have to be modified to mention the new operator. (This comes about because the decomposition of operators is stated in terms of states to be achieved, not directly in terms of the operators that achieve those states). Independence of operators, like modularity, facilitates change by limiting the impact on other operators.

With these considerations in mind, we turn to a number of examples, involving special cases of operators and failure recovery strategies.

5.1.1 Examples

Adding special case operators to a library may require that preconditions or subgoals of existing operators be rewritten. For example, when testing a system that is intended to fix certain bugs, the programmer should run the official testcases associated with those bugs, in addition to those testcases that would otherwise be selected. One solution (that keeps testing considerations local to the set of testing operators) is to write a separate operator covering all testing needed when bugs are being fixed; this operator has an extra subgoal and its precondition restricts its applicability to systems intended to fix bugs (this precondition might be an extended state predicate, as described in Chapter 3). Then it is necessary to prevent the normal testing operator, which has fewer subgoals and therefore does less testing, from being applied when this special operator is called for. Therefore, the normal operator for testing must be modified so that its precondition guarantees that it cannot be applied in cases where bugs are being fixed*.

To accommodate other types of special cases, existing operators may have to be rewritten in artificial ways. Consider testing a system that is about to be released to a customer; such testing should include running the testcases in the regression test suite (again, in additional to normally selected testcases). This would normally be done with a special *test-for-realese* operator with an extra subgoal; the precondition of *test-for-release* concerns the intent to do a release, i.e., the existence of a goal to release the system. But there is a problem expressing this precondition. There is no way to talk about the goals that

---

* One could institute a fixed preference strategy to select the operator with the most specific precondition that can be satisfied. However, in general this is overly restrictive — it would prevent a car buyer from financing the purchase by selling stock to raise funds because taking out a car loan is the most narrowly applicable operator.

are instantiated (i.e., expressions that are not now true but which are intended to be made true); it is only possible to talk about what expressions are true or false. In fact, there is no way to talk about any other aspect of the plan in which a given operator is actually instantiated. In the absence of a way to express the necessary precondition directly, some artificial means is needed to ensure that release testing is done exactly when it is needed

One recourse is to write separate operators with artificially different goals. This gives two operators, *test* and *test-for-release*, where the first has a goal of *tested(system)* and the second has a goal *release-tested(system)*. Then, operators (like *build*) that have testing subgoals will be affected. There will have to be two build operators: *build-for-release* will have a subgoal involving *release-tested* and *normal-build* will have a subgoal involving *tested*. This causes two problems. First, there is a proliferation in the number of operators. Second, it defeats the aim of encapsulating testing considerations within the testing operators.

(A slightly different solution, made possible by the fact that the differences between the normal and special cases are strictly additive, was used in the examples in Chapter 1. In operator library shown in Figure 1.3, the special requirements for regression (and performance) testing are captured in the precondition to the release operator. This solution does avoid proliferating special cases of the build activity. It still has the disadvantage that knowledge about how to test is not confined within the test operator(s).)

Expressing special cases with this brute force approach, already attended with disadvantages, breaks down entirely when multiple special cases affect a single operator; the combinatorics are intolerable. Special cases are not guaranteed to be simply additive with respect to the normal case. At worst, separate operators must be provided for all combinations of special factors.

In dealing with recovery from operator failure, there are problems both in connecting the right recovery operator with a failure situation, and in simply expressing the recovery strategy itself. Sometimes special operators are used for failure recovery, and only for failure recovery; for example, one of the actions for dealing with a compilation failure due to bugs in the compiler is to report the compiler bug. *Report-tool-bug* can be written as an operator, but how will such an operator get instantiated? There are no constructs to indicate what goals (and therefore what operators) should be instantiated when a failure occurs. At other times, the recovery strategy may involve executing some normal operator in a special way. If the *build* operator fails because the system being built is faulty (as would be the case if the linker detected errors), then one recovery strategy is to restart the build process using the faulty system as the baseline from which to edit. Expressing such a strategy requires access to the variable bindings of operator instances; again, this is beyond the scope of domain predicates.

5.1.2 Extending Representational Power

These problems have previously been tackled separately on a case-by-case basis, introducing special operator language constructs covering selected cases. The policies in [McDermott, 1978] represent one approach to the issue of matching special case operators to the appropriate circumstances. These policies derive their power from the fact that the NASL language allowed the writer to use plan-oriented constructs, such as <policy> IMPLIES (TO-DO <task> <operator>) or <policy> IMPLIES (RULE-OUT <operator>), in addition to domain-oriented constructs. Recovery from failure of operators is addressed in SIPE [Wilkins, 1985] which provides a special error recovery language. Domain-independent strategies, such as retrying a goal, are parameterized for use in operator-specific ways.

Finding a single formalism that extends representational power for a broad class of problems requires observing what the examples have in common. A compact way of describing the compiler bug example is: when there is an instance of the *compile* operator that has failed, build a plan to satisfy *tool-writer-notified-of-bug*. A compact way of describing the regression testing before release knowledge is: when there is an instance of the *test* operator appearing in a plan whose goal is releasing a system, add a subgoal *regression-tests-completed.* From these descriptions, two common characteristics emerge. First, there is commonality in the "domain of discourse": the entities being referred to are plans, their subparts (instances of operators, preconditions and subgoals), and their relationships and attributes (part of, failed, etc.). Second, there is commonality in the active form of the knowledge: it describes modifications to be made to these entities (e.g., satisfy a goal, add a subgoal).

This suggests that a general solution lies in facilities for expressing knowledge as transformations on plans. Transformations are, by definition, operations on some world state; in this case, the world state is the plan network. Therefore such transformations can be formalized as metaoperators and synthesized into metaplans. These metaoperators and metaplans capture domain knowledge, not control knowledge as in [Stefik,1980b] or domain-independent knowledge as in [Wilensky, 1980].

The primary advantage of this method is expressive generality, as compared with a collection of ad hoc operator language extensions. Any aspect of an operator definition (such as preconditions, subgoals, constraints, or effects), as well as any aspect of an operator instance (such as bindings of variables or ancestor operator instances) can be accessed or modified. The transformational approach also addresses some practical problems associated with developing and maintaining a complete library of operators. It helps to preserve the modularity and independence of operators that are otherwise sacrificed in the brute-force approach to expressing some of this domain knowledge. And, because

knowledge of exceptions is partitioned from knowledge of normal cases, the two issues can be tackled separately. The process of writing operators is further improved because multiple transformations can apply to a single operator, thereby preventing combinatorial explosion in numbers of operators.

In the remainder of this chapter, we develop this approach. First we look at examples of transformations in more detail. Then, we discuss the formal representation of transformations in metaoperators and metaplans. Finally, we discuss the implementation issues in this approach.

5.2 Transformations on Plans

Transformations apply to a plan network, the basic data structure of a planner or plan recognizer. Normally, operations on a plan network involve choosing operators to achieve states, instantiating these operators, and resolving conflicts between newly revealed states and existing states. Each such action can be thought of as taking the plan network one step closer to completeness. In contrast, a transformation will reformulate the current state of the network, with the effect of changing the solution set that will be pursued to complete the network. Such a reformulation is necessary either because the existing state of the network does not accurately reflect special circumstances or because other actions have reached a dead end (for example, an operator failure has occurred).

The examples in this chapter are based on the set of operators shown in Figure 5.1. They differ slightly from the example in Chapter 1, in response to the current concerns with modularity and information hiding.

5.2.1 Example: A Special Case

Software development is a domain where a large part of the knowledge about how actions are carried out is concerned with special cases. Consider a transformation that

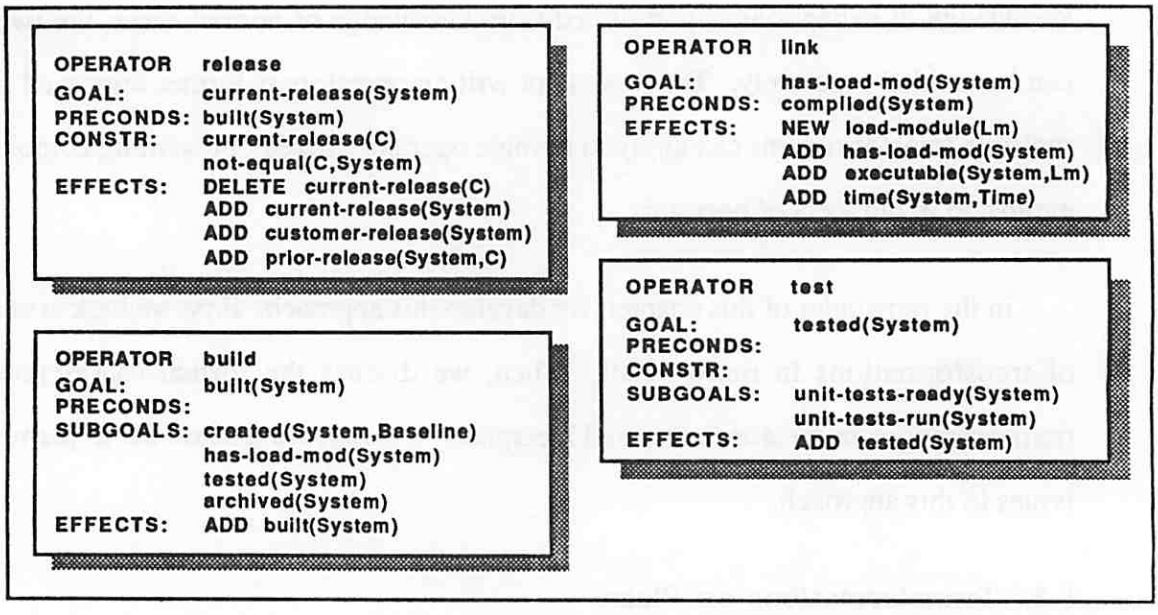

Figure 5.1 Revised Operators

implements the requirement to do regression testing before a release. When expressed precisely, the transformation affects an instance of *test* occurring as part of the expansion of *release*. To be entirely safe, one additional restriction should be given: that the system being tested is the same as the system being released. This will allow other testing instances to occur in the same expansion (such as running a testcase to help decide what editing changes are needed), while ensuring that regression testing is required on the right one. Expressing this condition requires access to the dynamic correspondence between the variable names used in the two operators. The BEFORE case of Figure 5.2 shows one situation in which this transformation is applicable.

Assuming the *test* operator of Figure 5.1, the effect of the transformation is to add an additional subgoal to run the regression test cases. The formula defining the new subgoal is supplied explicitly in the transformation—it need not have appeared previously in the plan network. This subgoal happens to be an iterated subgoal, and the iteration

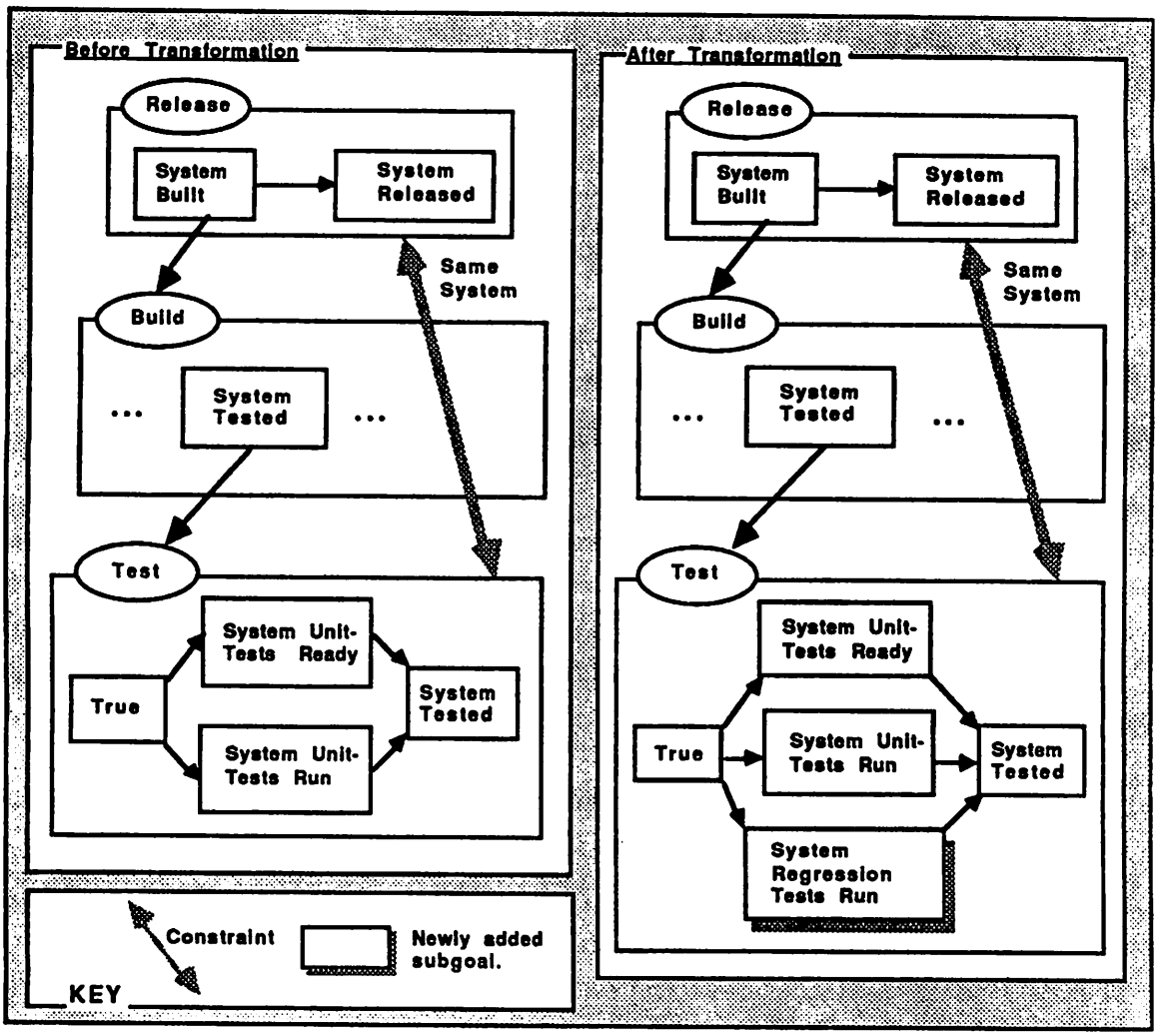

Figure 5.2 Transformation for Regression Testing

specification is also supplied explicitly in the transformation. Only the one operator instance is modified; the basic operator definition for test is unchanged. The results of applying this transformation are shown as the AFTER case in Figure 5.2.

5.2.2 Example: Failure Recovery

Software development is also a domain where there are many causes of failure, including system problems, tool problems and programmer error. In particular, given that

much work is carried out on a trial-and-error basis, failures due to programmer error are to be expected frequently. Consider the case of the *link* operator failing to produce a load module because errors made by the programmer were detected. In fact, decisions about recovery from this failure are not made at the local level of the *link* operator; a *link* operator failure implies that the parent operator has failed, and the appropriate recovery strategy is dictated by what that parent operator is.

Consider the case where the parent operator is the *build* operator. If the *build* operator has failed and no load module was produced, one recovery scenario is to go through the whole build process again; but, instead of starting from the same baseline used in the original *build* instance, this new *build* instance will start with the "faulty" system as the baseline. That is, the new *build* instantiation will use as the binding of its baseline variable the binding of the system variable from the failed *build* instantiation.

These strategies can be expressed in two separate transformations. The first transformation applies to instances of *link* that have failed; its effect is simply to mark the parent operator instance as failed. The second transformation applies to instances of *build* that have failed; its effect is to create a new instantiation of the *build* operator, and to fix the binding of the baseline variable in that instantiation to be equal to the binding of the system variable from the "superseded" instantiation of *build*. The effect of these two transformations is shown in Figure 5.3. Note that the original instantiation of *build* is no longer the "achiever" of a higher level state; it now "contributes" to achievement of that state.

5.2.3 Other Examples

The software development domain is particularly rich in examples that demonstrate the generality of the transformational approach. Some additional generic uses of

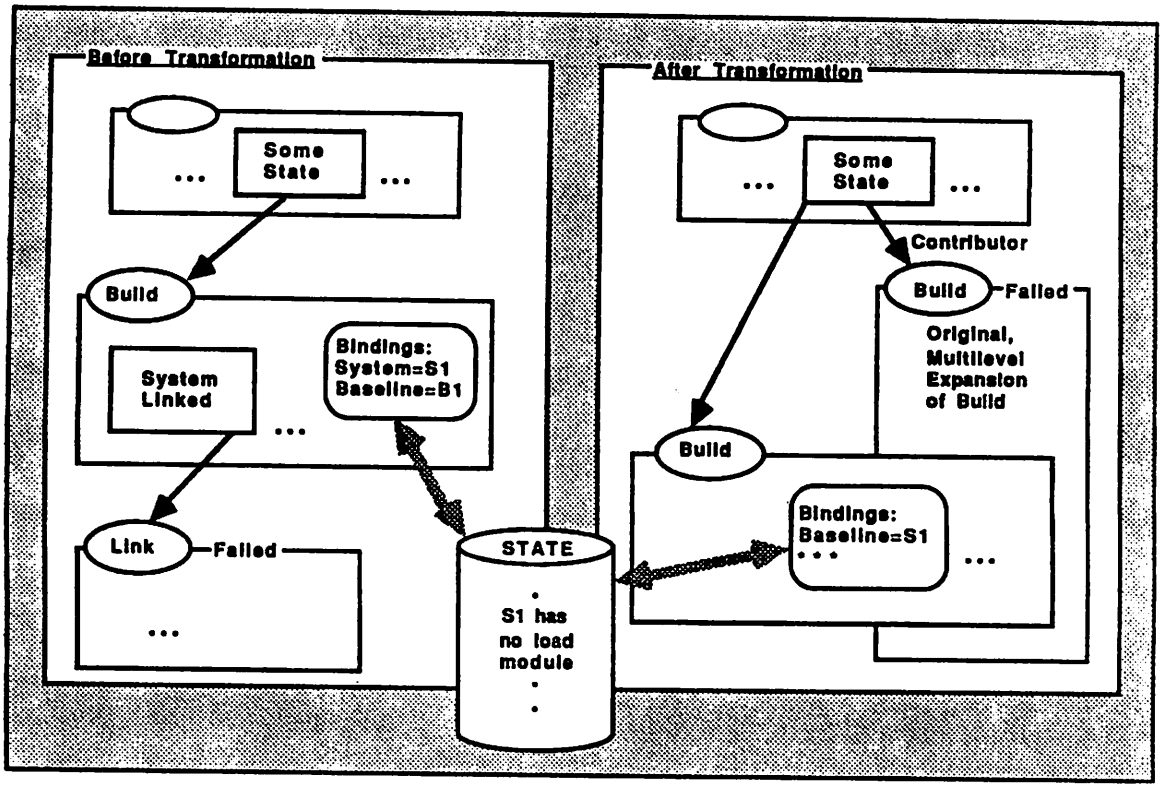

Figure 5.3 Transformations for Link and Build Failure

transformations, beyond representing special cases and straightforward failure recovery strategies, are described below.

5.2.3.1 Posting Goals

Transformations can be used to maintain desirable domain states in a flexible manner. (The NASL system used policies this way [McDermott, 1978]). That is, whenever an undesirable state obtains, a goal can be posted (i.e., instantiated) to reestablish the desired state. This is more forgiving than preventing the undesired state absolutely. (Prevention may be called for in some cases, while for others, reachieving the desired state is appropriate. Sophisticated prevention techniques for a planner are described in [Hogge, 1988].)

As an example, programmers generally follow a set of rules about how files are allocated to directories. However, in the heat of activity, a file may be created in the "wrong" directory. A transformation could trigger on this and instantiate a goal to move the file to the proper directory. (The precondition to the transformation is the fact that the undesirable domain state obtains, and the goal of the transformation is the existence of a user-goal to reestablish the desired domain state.) Once posted, the goal will show up on agendas of work to be done, to remind the programmer of an outstanding task.

Transformations that instantiate new top level goals work well in conjunction with focusing heuristics (described in section 2.2.2.3). An interpretation that involves satisfying an instantiated, but not yet addressed, top level goal ought to be preferred to an interpretation that generates a new top level goal. So, use of this type of transformation gives rise to an additional focusing heuristic.

5.2.3.2 Resolving Conflicts

One of the advantages of a planning system is the ability to detect and resolve conflicts in plans using domain-independent techniques, as discussed in section 1.3.3.2. However, there may be domain-dependent techniques as well, and these can be expressed as transformations. The precondition of such a transformation is that adverse interactions between two planned actions have been detected; the goal of the transformation is that the interactions have been resolved.

In the software domain, the operator that copies the contents of one file into another can be inserted into a plan to correct an interaction arising from parallel goals to modify some software object. (In this case, simply forcing one goal to be achieved after the other will not by itself resolve the conflict—the first goal must be carried out without destroying the original object.) Once there are two copies of the object, the interaction is resolved—each modification activity can proceed without interfering with the other. However, some

explicit clue is needed to ensure that the use of copy is considered among the solutions for interactions. That clue is provided by the transformation.

5.2.3.3 Shortcuts

Transformations can be used to apply shortcuts in just those situations where the shortcut is safe. A shortcut amounts to substituting one goal which is "easier" to achieve for another which is "harder" to achieve. The safety of the shortcut may involve the plan context in which the goal is instantiated. Thus, the metalevel constructs of the transformation are doubly necessary—first, to express the safety conditions and second, to describe the actual shortcut itself.

In the software domain, if editing a source module consists of cosmetic changes only, then an alternative to recompilation is simply to acquire (and place in the appropriate directory) the object module of the previous version (assuming no include modules were also changed). However, it is bad practice to do this on a release to a customer; in that case, it is generally advisable to recompile everything from source form to verify that the sources representing the release are in fact complete and correct. By expressing this shortcut in a transformation, we can ensure that good practice is followed.

5.2.3.4 Revising Goals

In some cases when operators fail, the recovery strategy may involve rephrasing the goal in order to proceed with the overall plan. In these cases, a failure indicates that the goal itself (in all its detail) cannot be accomplished; but there may be a related goal that will suffice. A transformation, whose precondition is that an operator failed to achieve its goal and whose effects are to substitute a different goal for the original goal, can express this strategy. A software example arises if the compiler blows up when operating with optimization enabled. A well established strategy is to try again with optimization disabled.

If this results in a successful compilation, the programmer will settle (at least temporarily) for a load module which is not optimized.

5.2.3.5 Generic Transformations

Transformations can apply to a specific operator (such as the test operator examples given earlier), or to any operator having a particular characteristic, such as a specific goal or subgoal formula. They can also apply to arbitrary characteristics of operator definitions, such as any operator having a particular type of parameter or parameter binding. In a multi-user system, when the number of users logged-in is below a certain threshold, then commands will be submitted for foreground execution rather than to a background queue. Suppose all operators representing CPU-intensive commands are written with an explicit parameter set for background execution. Then, one transformation, applying to any operator utilizing the background queue, can handle foreground/background selection. This transformation saves the author of operators from writing an additional version of each CPU-intensive operator.

5.2.3.6 Complex Transformations

The examples of transformations given so far are relatively simple (in formal terms, they involve a metaplan consisting of a goal achievable by a single metaoperator). However, transformations can be arbitrarily complex. In the software domain, recovering from failed operators entails deleting any extraneous files that may have been created before the failure occurred. One transformation could identify these files and instantiate goals to delete them. This transformation applies one change (instantiate a goal with specific variable bindings) many times (for each selected file).

Another complex transformation in the software domain applies to the conservative editing style of frequently saving a snapshot of the file being edited; here the intermediate

snapshots must eventually be deleted. This is a complex transformation involving two separate (but related) changes in the user's plan net: one change instantiates goals to save snapshots, and the other change instantiates goals to delete all but the desired version.

5.2.3.7 Sharing Operator Libraries

A final use of transformations is to allow sharing of a generic operator library among several applications, where each application has special requirements. In the software domain, several projects can share a generic operator library, if each expresses project-specific policies as transformations. Then, one project can require that a particular analysis tool be run before a customer release, without affecting whether other projects use the same tool in the same way. Separate sets of transformations are used to customize the operator library to the needs of each specific project.

5.3 Formalizing the Transformations

Because transformations are operations on the plan network, they can be formally represented as metaplans, synthesized from metaoperators. Metaplans have been used before in planning systems. Procedurally-implemented metaplans have been used [Stefik, 1980b] to pursue control issues in planning. Declarative metaplans have been used [Wilensky, 1980] in order to share (domain-independent) knowledge between a planner and a plan recognizer. Neither of these metaplan systems was used to modify operator instances by adding new subgoals, changing constraints, rephrasing goals, and so forth. Metaplans that could modify steps or change parameter bindings were defined for a natural language dialog understanding system [Litman, 1985]. In these metaplans, the modifications were metaplan parameters which were bound from information in the utterances.

In order to formalize transformations as metaoperators, it is necessary to define the metalevel state schema and to establish that the operator definition language has the necessary constructs to handle any special requirements of the transformational approach. Following this discussion, we give two examples of metaoperators.

5.3.1 The Metalevel State Schema

The metalevel state schema, covering the internal data structures used in planning, is given in Figure 5.4. The objects and relationships shown are representative, not exhaustive. As a state schema, Figure 5.4 is comparable to Figure 1.4; it is presented using the same ER model of data and consists of objects, their attributes and relationships. However, the domain of Figure 1.4 relates to the software process while the domain of Figure 5.4 relates to plans themselves.

The state schema in Figure 5.4 contains objects and predicates organized into three subspaces: operator library, plan network, and the domain state. The operator library subspace describes all components (preconditions, effects, etc.) of all operators, and their formulas and (static) variable names. The plan network subspace describes the hierarchy of operator instances: their dynamic status (started, completed, failed...), their internal components (preconditions and subgoals) and status (pending, achieved, protected...). Also associated with operator instances are the dynamic name scopes and variable binding information needed to evaluate operator formulas. The domain state represents the truth value of all domain predicates. Additional predicates cross subspace boundaries, relating operator instances back to their definitions, and operator instances to the domain predicates they affected.

In Figure 5.4, the operator library subspace and the plan network subspace are arranged to make parts of operators and instances of these parts into first class objects, rather than attributes of the objects they are part of. Thus, preconditions and subgoals and

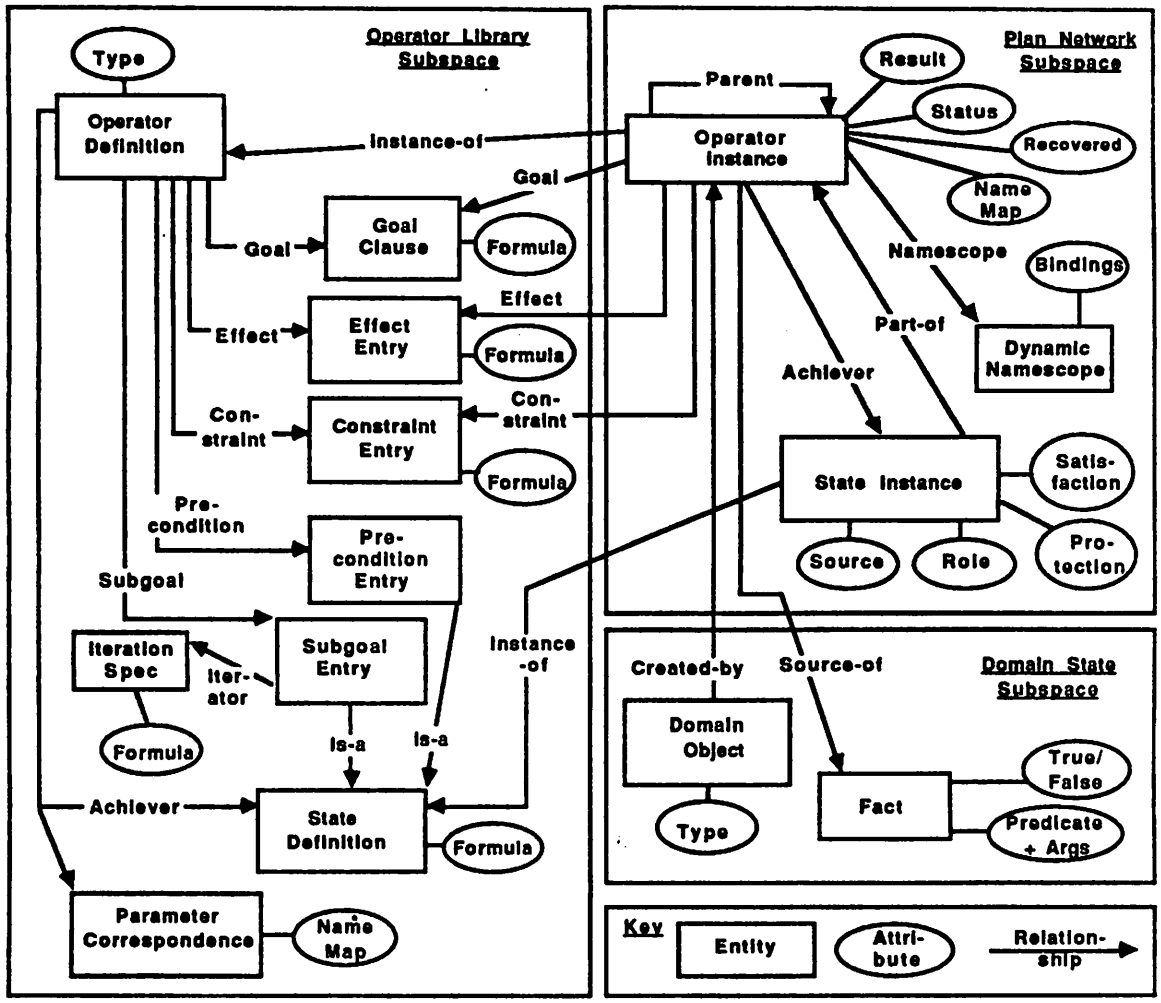

Figure 5.4 Metalevel State Schema

constraints, in particular, are objects with attributes and relationships in their own right. This facilitates making changes in both subspaces. To delete a subgoal from an operator instance, it is not necessary to make any changes in the library subspace (such as making a new operator definition with one fewer subgoal); one can simply delete the subgoal instance directly. And, in the case of adding a subgoal to an operator instance, the change in the library subspace only involves making a new subgoal entry (which is not part of any operator definition), not an entire new operator definition.

5.3.2 Metaoperator Constructs

Transformations often require complicated manipulation of the plan network (as the examples of section 5.2.3 show), so expressing them as operators requires a language engineered for "real-world" use. Clearly, effects of operators must be able to create new objects (for example, a new subgoal instance). Some transformations (such as the one to delete extraneous files) require a facility for iterating subgoals: that is, for repeatedly achieving a subgoal formula over a set of bindings. Computed predicates are also valuable; instead of maintaining explicit links between an operator instance and all of its ancestors in a plan, the ancestor relationship can be computed using the parent relationship.

Conveniently, these facilities are already present in the operator definition language described in section 2.1.2. However, one addition is useful: a keyword to distinguish between operators and metaoperators. (Since the two types of operators use different "domain" predicates, they can actually be distinguished without this keyword.)

The operator definition language that we have defined provides a keyword for distinguishing the top level operators (examples of usage appear in Appendix A). Top level operators are those that achieve top level goals, the goals at the highest level of abstraction in the plan hierarchy. In the case of metaoperators, the set of top level metagoals can be formed by collecting all the goals of top level metaoperators (and throwing out any duplicates.) The set of top level metagoals is used to determine when to apply transformations, as described in section 5.4.2.

5.3.3 Metaoperator Examples

The transformation for regression testing before release is shown as a metaoperator in Figure 5.5. This will be both a top level operator and a primitive one (it has no subgoals). The precondition will be true when there is an instance of *test* whose ancestor is an instance

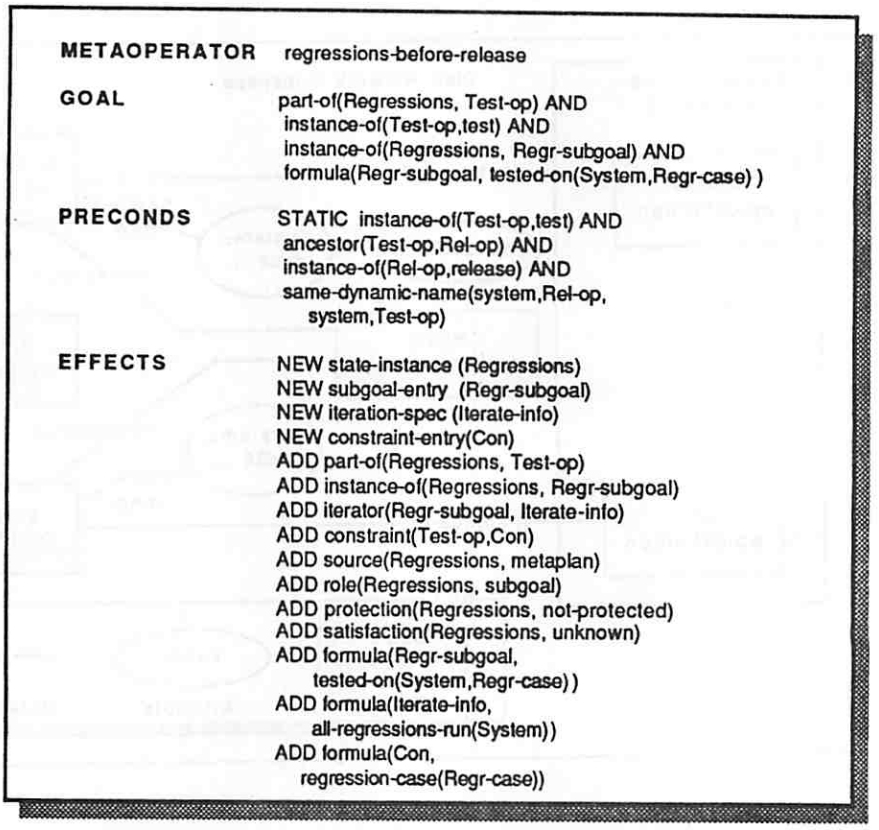

Figure 5.5 Metaoperator for Regression Testing

of *release* AND when the system variables of both operators correspond (this is determined by the computed predicate *same-dynamic-name*, which takes two static names and the operator instances they appear in, and determines whether they are both mapped to the same dynamic name). The goal is to have a state instance for regression testing in an instance of the *test* operator.

Figure 5.6 shows a case in which the transformation is applicable; it is essentially another presentation of the BEFORE case in Figure 5.2, showing more detail in the parts that are relevant to the transformation.

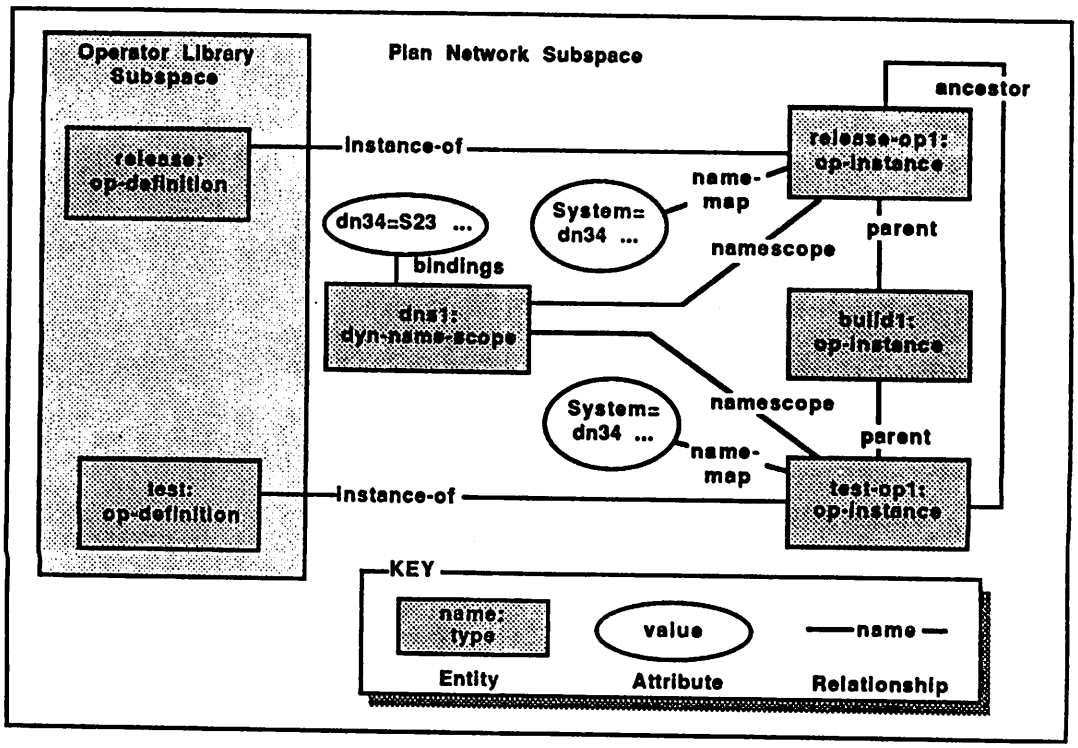

Figure 5.6
State before Application of Metaoperator for Regression Testing

Performing the transformation is simply a matter of realizing the effects of the metaoperator in this case, because there are no subgoals to be achieved. These effects are all directed at creating a new state instance as part of the *test* operator instance. The new state instance has a supporting subgoal entry that defines the state formula and, since it is an iterated subgoal, the iteration formula. Note that the metaoperator contains these formulas explicitly; they consist of domain predicates and variable names in the static name scope of the test operator definition. The transformation also adds a constraint to the *test* operator instance. This constraint ensures that only regression test cases can be used to satisfy the new subgoal. The state shown in Figure 5.6 becomes the state diagrammed in Figure 5.7 after the transformation is applied; note additions in both the operator library

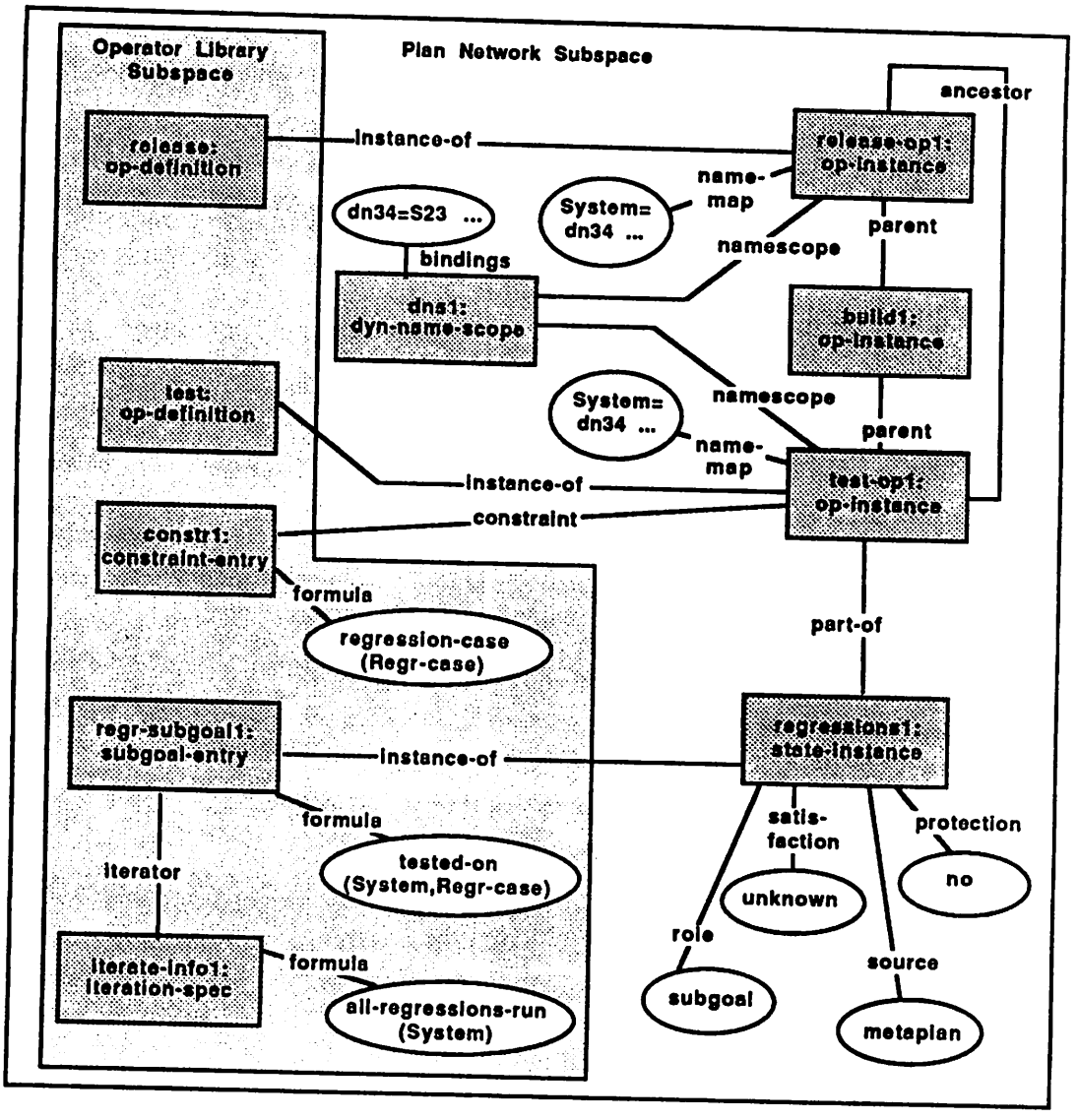

Figure 5.7
State after Application of Metaoperator for Regression Testing

subspace and the plan network subspace. (Figure 5.7 is a more detailed presentation of the relevant part of the AFTER case of Figure 5.2.)

The metaoperator for the regression testing transformation as given in Figure 5.5 is actually somewhat inefficient. It will make the same additions to the operator library subspace each time it is applied, thus replicating definition information that need only be

given once. The most attractive way to solve this is to allow definitions of fragments of operators as well as the definition of entire operators; this obviates the need for dynamic additions in the operator library subspace. Thus, as shown in Figure 5.8, in addition to the definition of the test operator with its subgoals for making and running unit tests, there would be a definition for a subgoal to run regression tests and a separate definition of the necessary constraint; these fragments belong to no operator definition. Then, the metaoperator for *regressions-before-release* only needs to instantiate this subgoal and this constraint in its effects—their definitions already exist.

(There are at least three other ways to handle this issue. The most direct is via the use of conditional effects (as described in section 2.1.2.7), so that those effects concerning the operator library subspace are conditional on no existing subgoal entry describing regression testing. Another alternative involves the use of three separate metaoperators. The first is a complex metaoperator with two subgoals: one involving adding the subgoal entry in the operator library subspace and one involving the instantiation of that subgoal entry as a new part of an instantiation of test. The other two metaoperators are primitive and are used to achieve the subgoals of the first metaoperator. On all applications of the transformation subsequent to the first application, one of the subgoals will already be satisfied and

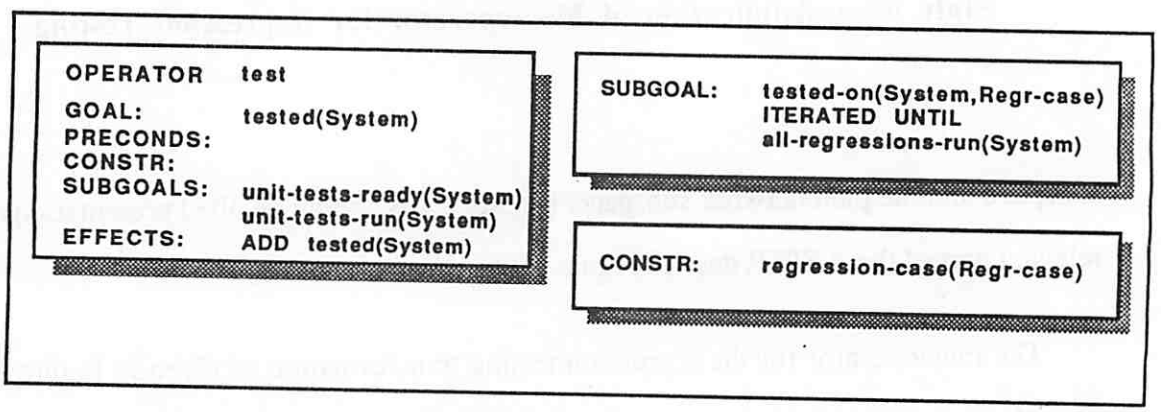

Figure 5.8 Defining Operators and Operator Fragments

metaplanning will ensure that actions to "reachieve" it are omitted. A final alternative requires exactly two metaoperators: one applicable when no subgoal entry for regression testing exists, and one applicable when that subgoal entry already exists; the effects in the second metaoperator are a subset of those in the first.)

The transformation exporting the failure of the *link* operator to its parent operator is shown in Figure 5.9, to show how a goal for failure recovery is expressed. The predicate *query* that appears in the precondition is a computed predicate that allows a domain predicate (here, *faulty(System)*) to be evaluated in the dynamic namescope of a particular operator instance (here, the failed *link* instance).

5.3.4 Role of a Macro Language

The detail in the preceding section may provoke concern about potential disadvantages to the transformational approach. Two problems are that writing metaoperators requires the writer to understand the internal data structures of the planning system (as well as domain complexities that are already challenge enough) and that improperly written metaoperators

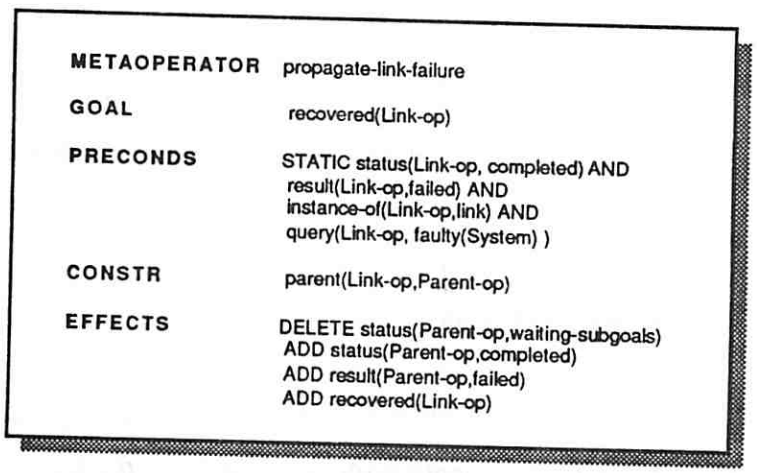

Figure 5.9 Metaoperator for Link Failure

can cause errors in the planning system operation by disturbing these data structures. These are real issues, but they have a common solution.

These apparent disadvantages can be overcome by providing a "macro language" in which to write metaoperators. For example, three macros to be used in metaoperator effects clauses would be add-subgoal, delete-subgoal, and modify-subgoal; each macro would have parameters identifying an operator instance and new/old goal expressions. The macro expansions handle all the details associated with each operation, and need not be written by (or understood by) the domain expert; they can be written by those who know the planning system implementation, and tested in advance for correctness. Macros are not only an advantage to the writer of metaoperators; they provide the planning system implementors with a way to separate the abstract data structure interface from its actual implementation. This allows change to the internal data structures without affecting the metaoperators—only the macro definitions need to be changed.

The important point is that a macro language is not an ad hoc set of features. *It is* backed by a theory of transformations that is very general, even though, at a given point in its development, the macro language may only implement part of that theory.

5.4 Implementation Issues

In this section we describe the issues in implementing the transformational approach in a plan recognizer.

5.4.1 Multiple Interpretations

In plan recognition, as we have already discussed in section 2.2.1.2, there will be multiple interpretations that compete with one another. Each of these interpretations consists of a plan network and a domain state. Thus the plan network and domain state subspaces as described in Figure 5.4 will have multiple instantiations. And, clearly,

different transformations will have been applied, or will be applicable, to different interpretations.

On the other hand, there is only one operator library corresponding to the library subspace of Figure 5.4. This subspace is an internal representation of all the information given with a set of operator definitions (such as those in Appendix A), to which is added the information generated by calculating the achiever relationships and associated parameter maps (discussed in section 2.2.2.1). Assuming the original operator library contains both complete operators and fragments of operators, then the operator library does not change dynamically. (If transformations such as the one in Figure 5.5 include changes to the operator library subspace, then these changes should always be formulated in a non-destructive way. This is achieved by expressing the changes as additions, so that that existing information is never lost. This ensures that a single operator library subspace can be shared by all interpretations.)

5.4.2 When to Apply Transformations

Reformulating plans via transformations represents a permanently instantiated objective of the plan recognizer. Thus, the execution of (top level) transformations will be data-driven: they will be applied whenever there is an opportunity to do so. This is done by looking for parameter bindings such that the static preconditions on some top level metaoperator are true; if the goal of this metaoperator is now false, then there is an opportunity to perform this transformation. (If the goal of this metaoperator is already true, then there is no reason to apply it.) When these two conditions are met, metaplanning is invoked to expand and execute a metaplan having the selected metaoperator as the top level operator.

5.4.2.1 Data-directed Application

It is important to perform transformations as soon as they are applicable to an interpretation. Examples show clearly what the issues are. If a transformation involves deleting a subgoal, the transformation must be applied before any actions are attributed to achieving that subgoal. And, the earlier the transformation is applied, the earlier the operator with the deleted subgoal will finish (and have its effects posted.) On the other hand, if a transformation involves adding a subgoal to an operator, the transformation must be applied before that operator is allowed to finish—otherwise, that operator will be allowed to finish prematurely. Finally, if a transformation involves deleting a constraint, then this must be done before the constraint is actually checked in any way.

New transformations may become applicable anytime during that part of the plan recognition algorithm that updates the status of the plan network (the last major segment of the algorithm shown in Appendix B). During this time, effects of operators are posted, which changes the domain state. Also, the plan network is changed as preconditions/subgoals are marked completed, failures of operators are noted, and new subgoals are instantiated. Any of these activities could trigger a transformation.

Transformations may also be applicable during generation of candidate interpretations. This causes a change in the strategy from the original algorithm. The original algorithm (Appendix B) for plan recognition delayed instantiating new interpretations until a path accounting for the new action had been both formed and then checked semantically. Paths were represented in a temporary data structure while they were being formed and checked; only if a path survived all checks was it then instantiated (using operator, subgoal, and precondition instances following the state schema in Figure 5.4) in a new interpretation.

Because paths and their viability can be affected by transformations, it is necessary to instantiate an interpretation including a new path as soon as that new path is formed. This

serves two objectives. First, it makes it possible to apply transformations, because the data structures that the transformations actually operate on are being used. Second, it ensures that transformations occur before the semantic checks are undertaken. For example, if a path involves an operator O and there is a transformation that deletes a constraint from O, then the transformation must be applied before the constraint is checked. (Note that if a transformation destroys a path, for example by deleting a subgoal on the path, then the candidate interpretation should be rejected.)

5.4.2.2 Goal-directed Application

In some special cases, transformations may have to be applied during the generation of candidate interpretations in order to create candidates, rather than in order to check them. This is best shown with an example. Consider the regression testing transformation, and assume that the first testing action to be taken involves running a regression test. Recall that new operators are only instantiated in a plan in order to account for new actions. Thus, the *test* operator is not actually instantiated until that operator is needed to explain actions. Figure 5.10 shows, at the left, the situation at the point of the first testing related action. In order to account for a regression test action, it is necessary to force instantiation of the *test* operator; only then can the transformation to add the extra subgoal be applied; the two steps of instantiating the *test* operator and applying the transformation are shown in Figure 5.10 in the middle and right respectively. All this must be done before it is possible to identify a path that will account for running a regression test case.

This extra work, creating an opportunity to apply a transformation, is necessary when no other path for the action exists. In other cases, this extra work may lead to generating an additional path (e.g., an additional candidate interpretation) for an action. Thus, it is always necessary to look for these opportunities, and hence this must be done efficiently. A data-directed approach, to look for all expansion possibilities, and then look for each

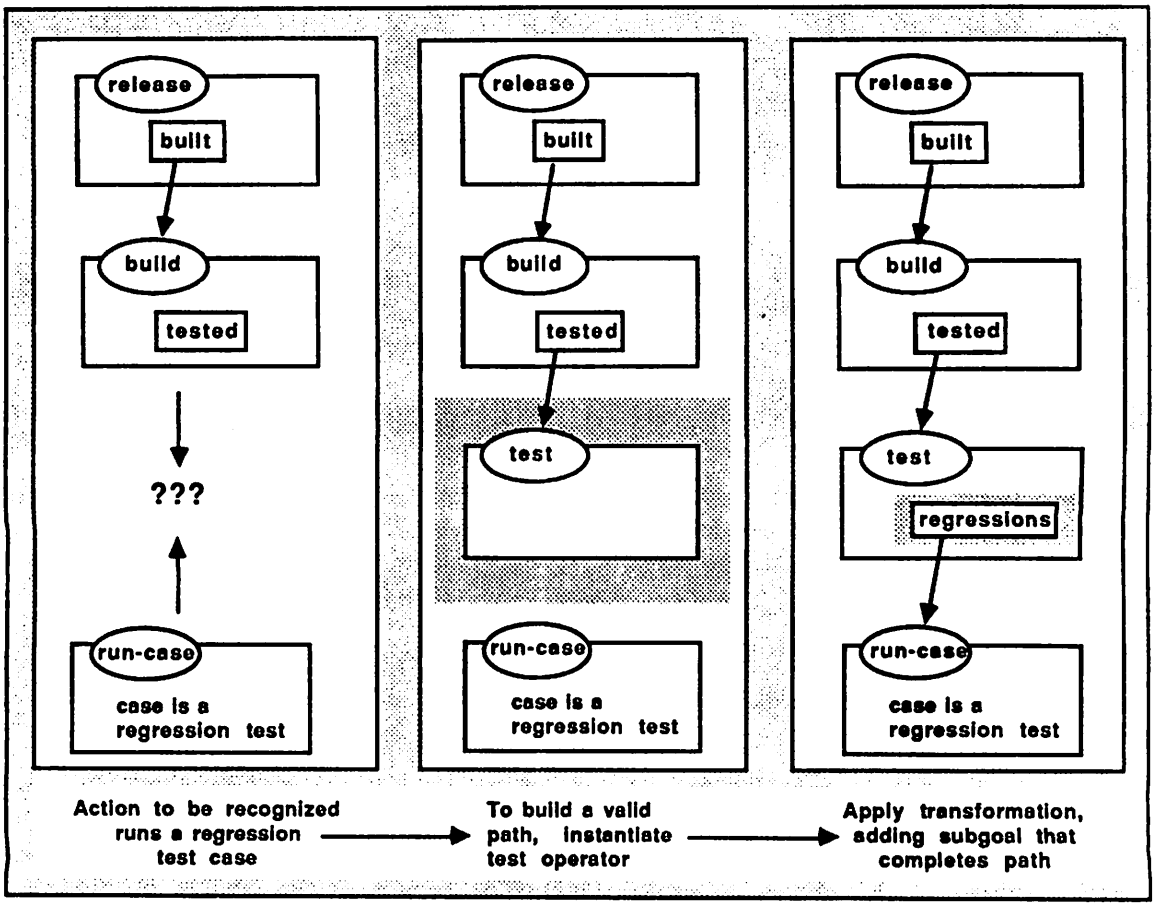

Figure 5.10
Applying a Transformation to Generate a Candidate Interpretation

transformational opportunity in each of these alternative expansions, can be time-consuming.

Efficiency here suggests a goal-directed approach to applying transformations. That is, a transformation is chosen not because it is currently applicable; rather it is chosen for certain characteristics of its effects clause, and then steps are taken to make the transformation applicable. (It happens that this also provides additional motivation to instantiate paths promptly rather than using the temporary data structure as is implied in Appendix B.) The goal-directed approach to applying transformations works generally as

follows. (The reader may first wish to review the section of Appendix B that describes how reachability information is used to generate paths in the normal case.)

First, identify all preconditions or subgoals that belong to no operator and that are reachable via a path from the current action. These are the missing steps in paths that have to be filled in dynamically via transformations. (If they are predefined, then their achievers have already been identified at the time the operator library definition was processed, and thus the reachability information is available). Then identify all metaoperators that instantiate such preconditions or subgoals. These are the metaoperators that we want to be able to apply. Now consider which (domain) operators are given new preconditions or subgoals by these metaoperators. These are the operators we have to instantiate if possible. Finally, identify the paths from the existing plans, or from a new top level plan, to these operators. Consider each such path; if no transformation is applicable to the path, reject it. Otherwise, perform the transformation, and consider paths from the current action that connect into this path.

In the regression testing example, this goal-directed process traces from the regression testing action to the regression testing subgoal through the reachability information. Then we find that it is the *regressions-before-release* metaoperator that instantiates this subgoal. From the effects clause of *regressions-before-release*, we see that it is the *test* operator that is given a new subgoal. Using the reachability information again, we see that *test* is reachable from a subgoal of *build*, which is already instantiated (as in Figure 5.10, left.) Then we instantiate the path from *build* to *test* (this happens to be a one-step path as shown in Figure 5.10, middle), after which the transformation is applicable. This creates the subgoal, which allows completion of the path between the action and the *build* operator (Figure 5.10, right).

5.4.3 Coordinating Transformations with Other Tasks

In plan recognition without metaplans, various operations of the plan network always occur at predictable times. This is no longer the case when metaplans are introduced, and some coordination between the transformations and other processing may be required. We give some examples of situations that require coordination.

In the absence of transformations, new operators are created only during generation of paths, and all name mapping is done at that time. With the introduction of transformations, name mapping has to be called at another time as well: whenever an operator dynamically acquires a new precondition or subgoal via a transformation. The precondition or subgoal could have introduced parameters that did not appear in the original operator, and these parameters must be assigned dynamic names in the appropriate namescope. For example, the regression testing subgoal instantiated dynamically by *regressions-before-release* adds a new parameter, *regr-case*, to the *test* operator.

We have already mentioned one case where applying a transformation may create extra work for the plan recognizer. If a transformation applies to the path accounting for the current action, then the path may be destroyed by the transformation. Thus, after transformations are made on a path, the path must be checked to see that it still provides a complete set of links between the top level and the action. This simple check can be done as the semantic checks are applied to a path (a process that traverses the path from bottom to top): if the path stops before the top level is reached, then it has to be rejected.

5.4.4 Completing Metaplans

We have to consider whether metaplanning will always be able to complete a plan to perform a transformation. Metaplanning is invoked when there is a top level metaoperator whose static precondition is true and whose goal is false. If that metaoperator has a

subgoal S, then another metaoperator MO needs to be selected to achieve S. It may happen that there is no MO whose goal achieves S and whose static preconditions are now true. Then, there is no way to finish metaplanning; however, at a later time, the static precondition for some MO may be satisfied, and metaplanning can be completed then.

This has two implications. First, in addition to looking for opportunities for invoking metaplanning on a new top level metagoal, it is necessary to look for opportunities to complete metaplanning on existing, but incomplete, metaplans. Second, in keeping with the need to apply transformations as soon as possible, that part of the metaplan that is finished should be executed even though it is not possible to generate or execute the remainder of the metaplan. This actually introduces additional complexities: what if the executed part of the metaplan is such that the metaplan will never complete? There is no certain way to recognize this—the transformation will forever be half executed.

An example of a metaplan that cannot be generated and executed in *toto* is the transformation for the conservative editing style, described in section 5.2.3.6. The (iterated) metasubgoal involving adding a subgoal to make a backup copy can be performed after each new editing action, but no metasubgoals involving adding subgoals to delete backups can be performed until all editing is done. It is possible to formulate this transformation as multiple, independent transformations, although this may require introducing additional domain predicates in order to achieve communication between these independent transformations.

We believe that an initial implementation of a transformational approach could assume, without any practical reduction in power, that transformations (i.e., metaplans) can be both completely designed and completely executed at one time. The transformation examples that we are aware of are largely straightforward from a planning viewpoint, and the few that are not can be split into simpler transformations without great difficulty.

5.4.5 Competing Metaplans

It will sometimes happen that there are two or more metaplans that can be created to achieve the same metalevel goal. For example, it could easily happen that there are two different error recovery strategies for handling one particular type of failure. The differences would be reflected in two different metaoperators that achieve the same metagoal. It is also possible to have competing metaplans that differ in parameter bindings, or in both metaoperator selection and parameter bindings.

Competing metaplans need to be handled by replicating the interpretation in which they apply so that each alternative metaplan can be executed on a separate copy of the original interpretation. Thus if two competing metaplans apply in interpretation A, it is replaced by two interpretations: A1, in which the first metaplan has been applied, and A2, in which the second metaplan has been applied. The new interpretations A1 and A2 compete with one another in the usual way, and differ only at to the effects of the metaplan applied. If the two competing metaplans represented two possible error recovery strategies, then the future actions of the user will be consistent with either A1 or A2.

This introduces the need for additional focusing heuristics to control search in the face of this proliferation of interpretations. Since interpretations can now differ as to the metaoperators applied, heuristics that distinguish on this basis are called for. Luckily, such heuristics are readily available. For example, one error recovery strategy may be more likely than another, and this heuristic knowledge could be used to focus attention, say, on interpretation A1 in preference to A2. Thus, the list of metaoperators applied to a given context is made visible at the focusing level, on a par with information about whether a context required planning, what its credibility was, and whether it involves starting new plans or continuing existing ones.

5.4.6 Optional Metalevel Goals

We have so far treated all metalevel goals as mandatory. Whenever there is a set of parameter bindings such that a metalevel goal is false and there is a metaoperator that is applicable, then a metaplan is created and executed. Execution of a metaplan makes permanent changes—the interpretation that existed before the metaplan was executed no longer exists. (If several metaplans apply, then several new interpretations are created, and the original one is deleted.)

Some metalevel goals, however, may be "optional". For example, a shortcut represents a course of action that a user is likely to take, but the user is not absolutely required to take a shortcut. In such a case, we want to create a new interpretation with the shortcut installed and treat the original interpretation as an alternative that is still viable. This can easily be done if there is an additional keyword, with value *mandatory* or *optional*, in the definitions of top level metaoperators. The value of this keyword determines whether the interpretation before application of the metaplan (that has this metaoperator at its top level) is retained or deleted.

Again, it is possible to create additional focusing heuristics that capitalize on this. For example, in the case of a shortcut, one might want to prefer the interpretation with the shortcut to the interpretation without it. This is expressed as a focusing heuristic that prefers the interpretation in which a specific metaoperator (here an optional one) has been applied to a interpretation without that metaoperator.

5.5 Summary

In this chapter, we have presented a general solution to a class of representation problems in planning formalisms. We have developed numerous examples of domain knowledge that cannot readily be expressed in standard operator format. While each of

these cases could be accommodated by designing additional features in the operator definition language, as was done in Chapter 2, a more general solution, based on expressing domain knowledge as transformations on plans, has been advanced. We have given examples to show the broad applicability of the transformational approach and have shown how these transformations can be formalized in metaoperators and metaplans. Finally, we have discussed the issues of implementing the metaplans in a plan recognizer.

This chapter completes the description of research contributions; in the next chapter, we summarize the dissertation and describe directions for future research.

CHAPTER 6
CONCLUSIONS AND FUTURE DIRECTIONS

In this final chapter, we summarize the results of this dissertation, provide some conclusions about progress to date and challenges that remain, and describe future directions.

6.1 Summary of Research

In this dissertation, we have described the need for a new type of user assistance, based on a *process* perspective towards computer-based activities. Instead of aiming solely for increased levels of automation in process support, we have placed primary emphasis on finding a comprehensive representation for process knowledge—one that will support various forms of assistance, including but not limited to automation, across all process activities. This support is provided by an *intelligent assistant* that is an active agent in the environment.

Our architecture for intelligent assistance is based on the integration of *planning* and *plan recognition,* complementary AI approaches to reasoning about actions. The key aspects of this architecture include domain-independence, mixed initiative support at multiple levels of abstraction, and explicit representation of goals. The intelligent assistant

is thus able to formulate plans that take advantage of what the user has already accomplished, without interfering with his other ongoing activities. This active assistance is provided via planning. Since the intelligent assistant can reason about the rationale and outcomes of actions, advice can be offered when actions taken by the user appear inconsistent with previous actions and known goals. This passive assistance is provided through plan recognition.

We have shown that achieving substantive assistance requires extensive domain knowledge, and have proposed as key issues the questions: what kinds of domain knowledge are needed, what representational limits are encountered in a planning formalism, and how can these limits be overcome? In answering these questions, we have shown that a planning approach already provides a certain level of built-in knowledge about actions and how to construct plans from actions tailored to the current context. Central to the planning approach is the data structure called a plan, which gives detail both about the structure and rationale for actions and about the significance of what is true in the current state of the world.

The investigation into representational limits has led to the identification of two limitations and the development of extensions to address each one. Before addressing any issues of limitations, it is important first to establish the foundation on which extensions can be built. We have done so by describing both a basic operator definition language and a plan recognition algorithm, each suited to the requirements of the intelligent assistant application. In the operator definition language, we gave some engineering extensions that increase representational power without raising any new research issues. In the plan recognition algorithm, we have solved a set of technical obstacles via the use of a planner to satisfy plan recognition objectives.

The first limitation we describe is due to the fact that the intelligent assistance application involves, by its very nature, hidden state—information about the state of the world is incomplete. In this case, it becomes impossible to evaluate preconditions, subgoals and constraints in operator definitions. Domain knowledge that would be applicable to doing this cannot be exploited because of the missing information. The very ability to model the domain in a nontrivial way is at risk. The solution to acquiring the missing information is to make plausible assumptions using empirical knowledge about what is typical or expected, rather than certain, in the current context. This reasoning can be formalized in a nonmonotonic system. The empirical knowledge allows fundamental domain principles to be exploited, so that reasoning about actions takes place in a deeper context.

We have given both the conceptual and technical details of incorporating this type of reasoning into a plan recognition system. The introduction of assumptions that may later prove incorrect affects how alternative interpretations are generated and compared. Credibility, the degree of agreement with current assumptions, is computed as alternatives are generated and becomes the basis for preferring some alternatives to others. Additionally, the plan recognizer must be able to detect when its current assumptions are (possibly) wrong, and be able to revise the assumptions accordingly. A new process, called reconciliation, is provided for revising assumptions; its objective is to fully integrate new information into the nonmonotonic reasoning by both propagating the consequences of adopting the correct assumptions and as well as providing a rationale for those assumptions.

The second limitation we have addressed concerns knowledge about the domain that directly concerns plans, not actions. This knowledge naturally lends itself to a transformational representation that allows for operations like substituting one goal for another, adding subgoals, deleting subgoals, and the like. Since these transformations act

upon plans as the state of the world, they can be formalized as metaoperators and synthesized into metaplans. This not only provides a single formalism in lieu of adding many special purpose constructs to the operator definition language, but it also extends the role of metaplanning in planning systems.

6.2 Conclusions

In this section, we present some overall conclusions, in the spirit of reflecting on where we are and where we are going. We assess the magnitude and difficulty of supporting software processes, the current state of progress, and a prognosis for the future. We also consider the difficulty of providing intelligent assistance (in any domain) and the appropriateness of a planning paradigm. Finally, we comment on the direction of nonmonotonic reasoning research.

6.2.1 Software Processes Are Complex

We believe that the examples given throughout this dissertation show that software processes are very complex entities. Knowledge about processes has many facets. No single approach will capture them all—multiple representations will be needed. We have shown four levels of representation, one each for: domain-independent knowledge about actions generally (embodied in planning algorithms), domain-dependent knowledge about specific actions (represented in operator definitions), knowledge that can approximate missing information (captured in nonmonotonic rules), and metalevel, domain-specific knowledge (expressed in metaoperators.) Others are discussed in section 6.3 as future directions.

Since software processes are so complex, we believe that there is no off-the-shelf paradigm that will serve in its present form. Every candidate paradigm will need to be extended in various ways, much the way we have found that planning technology needs to

be extended. Again, from our planning experience, some extensions can be expected to require new, basic research. This need for extensions will affect every candidate paradigm—rule-based systems, programming languages, database approaches, or state machines.

6.2.2 Process Support Is in Its Infancy

Processes have become a subject of attention only recently: an initial workshop on software processes was held in 1984 [Potts, 1984]; three others have followed (see [Wileden & Dowson, 1986], [Dowson, 1987], and [Tully, 1989]) and a fourth is about to be held. Active environments [Balzer, 1987], the first efforts to explicitly deal with process notions, are themselves very recent. And, as we have already mentioned, while there is general agreement that processes exist, there is currently relatively little understanding about what processes actually are. Clearly, the field is in its infancy.

Given the complexity of software processes, comprehensive representation of and support for software processes has to be recognized as an undertaking of great magnitude. Without entering into any debate about which is harder, it is at least possible to say that the magnitude is similar to that of automatic programming, which has been an area of active research for over 20 years. We note that while much progress has been made, automatic programming has proved to be remarkably difficult to achieve. However, given the advances that have been made in many fields (in software engineering but especially in AI), much more supporting technology is in place; this should accelerate pace of work on software processes, at least initially.

6.2.3 Intelligent Assistance Is a Challenge

Independent of considerations of the domain to be supported, intelligent assistance is by itself a difficult goal. To provide even simple advice requires extensive knowledge. As we have shown, that knowledge must be directed not just at "what" to do but "why";

otherwise there is no basis for adjusting actions to context, especially to exceptional situations. An assistant that is armed just with a set of directions (turn left at the third traffic light after the sign for Timbuktu, etc.) will break down on encountering a road closure; the "intelligent" assistant must be provided with a map, and the knowledge to reason from that map.

6.2.4 Planning Is an Appropriate Foundation

Planning as a technology has been a subject of study for about 20 years; another way to put this is that planning technology represents 20 years of studying actions from a general, domain-independent perspective. Operator definitions, formulated as goals-preconditions-subgoals-effects, have proved remarkably durable, with little substantive change in the basics of the approach during this time. (On the other hand, it must be admitted that planning has not seen the kind of massive technology transfer into real-world applications that has occurred for expert systems.)

The single key aspect of a plan-based approach is the explicit representation of goals. This is the primary feature that distinguishes planning from the alternative approaches based on programming languages or state machines (finite-state, petri net, etc.). The use of goals gives the system a deeper representation of actions, by explicitly showing their purpose. This becomes the means by which actions can be tailored to the current context (e.g., omitted because the goal is already satisfied) and the means by which potential conflicts among actions (e.g., destruction of a subgoal before its siblings are also achieved) are both detected and resolved. Furthermore, without goals, it is hard to see how the knowledge that we have been able to capture so readily in metaoperators can be handled.

Because of the explicit representation of goals, planning systems have two advantages. First, they already provide solutions to a collection of problems that affect processes—exactly those problems that are key to providing intelligent assistance. This means that,

compared to other paradigms, there are fewer additional problems to tackle initially. Second, planning systems are likely to be easier to extend. That is, extensions are likely to be just that—extensions—not major shifts in paradigm. When the foundation is right, extensions fit naturally; when the foundation is not quite right, extensions require that the foundation itself be reworked.

6.2.5 Nonmonotonic Reasoning Needs a Practical Orientation

Nonmonotonic reasoning is an important topic within AI, and one with many potential applications. Incomplete or noisy data characterize most AI problems, and thus there is both a need to make assumptions to proceed with problem solving and a need to revise those assumptions as more data becomes available. More progress has been made in nonmonotonic reasoning at a theoretical level than at a practical level; there is a need for more work in actually integrating nonmonotonic reasoning into problem solving systems.

Currently, there is considerable activity of a very formal nature, both in development and extension of logical systems for nonmonotonic reasoning (with circumscription as the current favorite) and in investigations into the relative power or formal equivalence of alternative approaches. Only recently have tractable algorithms for these approaches started to appear (see for example [Etherington, 1987] for default logic algorithms). Research has tended to focus on nonmonotonic reasoning in isolation, not in a particular problem solving application. The single exception is formalizing multiple inheritance systems, which has received a great deal of attention. In contrast, the TMS approach to nonmonotonic reasoning has seen wide application in problem-solving systems (usually as an approach to implementing search), but less attention at the formal level.

We have found that integrating nonmonotonic reasoning into a problem solving system raises many interesting issues that must be confronted. For example, when an nonmonotonic system leads to multiple interpretations of the current state of the world

(multiple extensions in default logic terms or multiple labellings in TMS terms), a problem solver will need to have ways to select among them; "resolution" by random selection merely begs the issue. Some solution must be provided, whether it is regarded as part of the nonmonotonic system, part of the application problem solver, or part of the interface between the two. We have proposed a metalevel facility (preferences) for doing this. Another example is that the problem solver may very well want to know which conclusions represent certainties and which represent assumptions; the plan recognizer used this information both in computing credibilities and in determining how to update the current state of the plans. In response to this, the nonmonotonic system must be able to make these distinctions in reporting its conclusions, as we have done via a simple five-valued logic.

Looking at nonmonotonic reasoning from different application perspectives can clearly suggest directions for further research. Our application of nonmonotonic reasoning to handle the hidden state problem has led to a novel approach to integrating new information with old information. This type of feedback from the application perspective is important if nonmonotonic reasoning is to serve a useful function in AI systems.

6.3 Future Directions

In this section, we discuss directions for future research. These are divided into three categories: software engineering, intelligent assistance, and AI.

6.3.1 Software Engineering Directions

Research directions that are primarily concerned with software engineering issues are empirical studies of software processes, trial application of intelligent assistance, and integration into a knowledge-based product environment.

6.3.1.1 Empirical Studies of Software Processes

In order to gain a better understanding of the organization and content of software processes, empirical studies are necessary. These can be conducted both by capturing programmer's command streams (much like the UNIX history mechanism does) and by interviews with programmers. The interviews are needed for two reasons. First, they will help to get a better understanding of the purpose of observed actions, which is not always obvious (as we have found from informal perusal of history logs). Second, they will fill in other details of processes that may not be observed unless a very large number of commands are sampled.

While the results of these studies could be presented in words, it is possible to capture the process knowledge which is elicited in the studies using all the knowledge representation techniques developed in this dissertation, i.e., in operators, nonmonotonic rules, and metaoperators. This has the advantage of ensuring a degree of formality in process understanding that is not possible with English language descriptions.

Empirical studies are also necessary in order to assess the effectiveness of the knowledge representation approach presented in this dissertation. Only with some empirical data on process knowledge is it possible to measure the coverage, and assess the gaps in coverage, of the representation techniques we have developed. In addition to assessing where we are, this will also suggest where to go from here. That is, the nature (and frequency of occurrence) of knowledge that cannot be formalized using these techniques will indicate the most important problems to be tackled and solved in extending the representation further.

6.3.1.2 Trial Application of Intelligent Assistance

We believe that the knowledge representation techniques presented in this dissertation provide sufficient coverage of process knowledge to make a trial application of intelligent assistance feasible (but that this would not have been feasible in the absence of the deeper domain modeling and metalevel knowledge). This effort should follow the empirical studies just described, which give a better understanding of process knowledge.

In order to carry out a trial application, it is necessary to select a particular process and create a formal process definition, i.e., to write a complete set of operators, justifications, and metaoperators (and perhaps identify some further focusing heuristics.) The magnitude of this undertaking should not be underestimated. Writing operators is very much like, and just as hard as, writing formal specifications. In particular, it requires the same shift in thinking towards "what to achieve" and away from "how to do it". As with writing formal specifications, however, speed and accuracy in formal definition improve with practice.

Before an actual trial in real-time with real users, there are multiple transition issues to deal with. Since these are largely domain independent, they are discussed below in section 6.3.2.1.

6.3.1.3 Integration in Knowledge-based Product Environment

The types of processes considered in this dissertation have been those typified by a generic UNIX environment. These processes have the advantage of being in actual use in both industrial and research settings, and therefore they are easily observed and studied. Study of existing processes is obviously the groundwork for developing processes of the future.

An interesting class of processes of the future are those that will be used in a knowledge-based product environment. These types of environments are still the subject

of active research; examples include the Programmer's Apprentice [Rich & Waters, 1988], KBRA [Czuchry & Harris, 1988], KBSA [Johnson, 1988], and CHI/REFINE [Smith et al., 1985]. Most research in this area is concentrated on the selection/development of appropriate knowledge representations and the design of tools (operations) to work with the knowledge. Attention has not yet been directed specifically to the processes in which these tools will operate.

Many aspects of existing processes will carry over into a knowledge-based product environment. Products will be not be monolithic. They will be divided into modules, and so notions of configurations of modules will be relevant. Products will not spring forth fully-formed. They will be developed incrementally, and so notions of history and previous versions will be relevant. Products will not meet their specifications initially. There will be various kinds of analysis to find deviations from specifications (whether by testing or executing a prototype or user-feedback on an executable specification), and products will be modified accordingly.

However, knowledge-based product environments do offer exciting new possibilities for processes, and therefore for process support. The major characteristic of these environments is that software products are not treated as text (or text with syntactic structure), but rather as objects with deeper structure. This opens up many opportunities at the process level.

Parts of the process, such as modifying an incremental version of a product, can be modeled at greater detail than before. The actions for doing this can be distinguished as to whether they add new functions, re-implement existing functions, elaborate an existing function, add error checking code, change the input/output conditions, etc. This makes it possible to model other aspects of processes with more detail. In a traditional process, it is difficult to model the actions taken within a text editor as part of a process—editing has to

be treated as an atomic action with no substructure; even if the editor is syntax directed, there is too little knowledge to work with.

6.3.2 Intelligent Assistance Directions

Research directions that are primarily concerned with intelligent assistance issues are transition to practical application, design of user-interfaces, role of knowledge acquisition, assisting multiple agents, modeling individual users, modeling user beliefs, and incorporating problem-solving styles.

6.3.2.1 Transition to Practical Application

Many issues stand between having demonstrated that plan-based intelligent assistance is theoretically feasible, and having such an assistant installed and running in a working environment. Before an idea is scaled up from small examples to comprehensive coverage of real situations, numerous transition problems must be confronted. Three issues which need to be addressed are dealing with large libraries of operators, attaining acceptable performance in the presence of complex plans, and tailoring the intelligent assistant to maximize the value of the assistance provided.

Before an operator library can be used in a working intelligent assistant, it must be "tested" or otherwise analyzed to ensure that the library in fact defines the desired process. A library of operators can be tested empirically by running plan recognition and planning on streams of commands captured from the actual working environment, prior to running plan recognition in real time. Errors and omissions in the process definition will be revealed and can be corrected. This undertaking has all the drawbacks of testing software—the difficulties of achieving coverage and estimating the prevalence of undiscovered problems.

There is a need for analysis tools to derive properties of the operator library and the process it defines; this gives a perspective orthogonal to testing and interesting in its own right. While analysis by human inspection is practical for small numbers of operators, it becomes infeasible as the number of operators increases. Analysis can range from simple issues like finding goals that have no achievers to more complex issues like proving that the process definition never allows the user to be in such-and-such a state or never allows doing X without doing Y. (Analysis of achievers and paths that can be constructed via the achiever links is already performed by the operator library preprocessor in GRAPPLE.) Proving properties of operator libraries and the plans they can generate has not really been addressed in previous work on planning, but is relevant to all applications of planning.

Performance of the plan-based intelligent assistant is a significant issue due to the computational complexity of the underlying algorithms (as described in section 2.2.5). It is not the size of the operator library *per se*, but rather the complexity and uncertainty in the plans generated, that increases costs of providing assistance. Performance measurement can be done, in advance of actually deploying an intelligent assistant, by running plan recognition and planning on input streams sampled from the real environment. These empirical measures of performance will be dependent on the domain and the characteristics of the given operator library. In addition to gathering statistics on the actual time required to recognize commands, it is also important to measure the effectiveness of the focusing heuristics, as described in section 2.2.5.2.2. New heuristics can be identified both by studying sample command sequences and by studying specifically those cases where plan recognition makes the wrong focusing choice during the test runs. If the best set of heuristics still does not lead to satisfactory focusing, then it may be necessary to solicit further discriminating information from the user, as suggested in section 2.2.5.2.2, or to include some user modeling as discussed in section 6.3.2.5 below.

The objective of intelligent assistance is to enhance the productivity of users of complex systems. There have been few empirical studies of the types of process errors that users make and the types of assistance that would be useful, and so there is little guidance as to what constitutes the best package of services. There are many interesting questions to answer. Will users take advantage of agendas and summaries? Should errors such as unsatisfied preconditions be corrected automatically or only after user confirmation? Should the assistant provide any types of automation without explicit user invocation? What degradation, if any, in response time will users tolerate to get intelligent assistance? How invasive can the assistant be (and when should it intrude on the user)? How tolerant will users be if the assistant offers the wrong advice (as for example, when the focus is wrong in plan recognition)?

Transcripts of commands from real environments (together with interviews of users) are one source of data. Another interesting source is the conversations that occur between a computer user and a colleague who happens to be looking on. We have observed that these interactions are rich in advice, both requested and volunteered; thus, they are good sources of what users actually need for support and the human standards against which an intelligent assistant will be measured. While informal study of this data will yield some guidance, carefully designed experiments to measure specific factors in a controlled way are clearly needed (and not a trivial undertaking). Feedback from such studies will surely influence how the interface between the intelligent assistant and the user is designed (this is discussed in more detail below) as well as the actual services provided by the intelligent assistant.

It is possible to simulate an automated intelligent assistant with a human mimicking what the automated assistant would do. This would allow some qualitative or quantitative measures, in terms of user acceptance and perceived productivity increases, of the value of

the assistance provided. In addition, it would allow for controlled experimentation with different types of support without having to build multiple prototype assistants.

6.3.2.2 User-interfaces for Plan-based Intelligent Assistance

In the plan-based architecture for intelligent assistance, three new conceptual entities have been introduced: goals, operators, and plans. The role of these entities in the user-interface has not been addressed. There are exciting possibilities both in the way the system communicates with the user, and the way users direct the system to perform actions.

Operator libraries and plans can be represented graphically in various ways (goals implicitly are included in both representations); one example presentation of an operator library was given in Figure 2.6 and two alternative presentations of plans appear in Figure 2.10. From these displays, users will want to retrieve both static information (what subgoals does this operator have?) and dynamic status (which subgoals are already achieved?). Some information can be encoded directly into the representation (different icons or colors for achieved and unachieved subgoals) and some can be presented on demand (for which a hypertext approach could be used).

An interface that allows for the display and querying of information about operators and plans is incorporated into the GRAPPLE system using the basic Macintosh conventions. For example, the windows for the operator library and current plans are scrollable, resizable, and movable on the screen. Clicking on components of these windows yields pop-up displays of additional detail; components can also be highlighted by the user. This interface, which was not designed with the end user in mind, could be adapted and then subjected to empirical studies of its effectiveness.

The most important implication of introducing goals, operators and plans affects how users actually direct the system to perform actions. If users can display their current plans, then they can identify goals to achieve or operators to execute by pointing and selecting components in one of these plans. During cooperative planning, users could supply their input to the planning process by *direct-manipulation* of these plans—expanding a subgoal with a new operator selection, filling in a parameter, etc.

Another issue in user-interfaces relates to the ability of a system to communicate on the user's terms rather than requiring the user to learn its terms. When the user does not know the exact operator or predicate names in use by the intelligent assistant, his ability to formulate a goal to be achieved or to understand an item on an agenda is limited. This issue has been addressed in UC, a natural language help system for UNIX [Wilensky et al., 1984]. UC represents knowledge about individual actions (as opposed to sequences of actions) and how users communicate about actions. It uses a frame-based structure for representing UNIX command knowledge that can be related to the linguistic content of a query. For example, in answering a user query about how to find out if two files are the same, UC must draw on knowledge that "same" is the opposite of "different" and that there is a command to find the differences between two files. The mechanisms of UC could be extended to encompass the process-oriented approach presented here.

6.3.2.3 Role of Knowledge Acquisition

Since process knowledge is complex and since operators are difficult to write, there is clearly a need to support the knowledge acquisition activities that must precede deployment of an intelligent assistant. One aspect of knowledge acquisition, the assimilation of new information into an existing library of operators, is treated in [Lefkowitz, 1987]. An approach to knowledge acquitision that emphasizes graphical representations for simulation and debugging of plans is given in [Mahling & Croft, 1988].

Given that one of the difficulties in writing operators lies in the use of a "what to achieve" perspective rather than a "what to do" perspective, it would seem that *learning from examples* is an attractive approach to pursue (a dynamic application of learning from examples, that results in the operator library being augmented during plan execution, is described in [Broverman & Croft, 1987]). The process expert would supply examples of processes, and the knowledge acquisition system would hypothesize interconnections among the actions, propose goals, ask for more detail on error conditions, etc., to construct and refine an operator library. This method is additionally attractive because there is some basis for building a model that explains how examples can reflect special, not general, situations. For example, actions may be omitted from examples because their goals were assumed to be satisfied, or two actions may described sequentially although their order of execution is not sequentially constrained; probing the example for these built-in assumptions and unintended implications will yield the missing detail. Also, when an example includes "if condition then do action", the condition may be the negation of the goal of the action, or may have to do with the possible failure of a previous action.

6.3.2.4 Assisting Multiple, Cooperating Agents

While limited aspects of the project context are reflected in the process followed by an individual programmer, a full treatment of coordination, cooperation and negotiation within the project requires modeling multiple, cooperating agents. Three examples of interactions at the project level are as follows:

- Cooperation: A new feature is to be implemented that involves modules authored by two different programmers. Assuming both programmers are available, each performs the modifications to "his" modules, and then one of the two programmers takes all the new modules, integrates them, and tests them.

- Negotiation: One programmer needs to make a change in a module that is currently checked-out with return privileges to another programmer. The options are for the second programmer to relinquish his rights to check-in a new version so that the first programmer can make the change, after which the second programmer can get the privilege back; for the second programmer to make the change and to check the new version back in promptly; or, for the first programmer to make the change with the intent of making it again after the second programmer has checked-in a new version. The options differ in timeliness as well as in work loading among the programmers.

- Coordination: One programmer is making changes to an interface that affects many other modules, authored by several other programmers. The changes can be made and tested locally, but their promotion to be visible to the other programmers requires careful timing. Advance notice of the intent to promote is advisable, not just for sake of courtesy, but so that others may arrange their work efficiently. For example, another programmer may choose this as a good time to fix some minor bug that will require recompilation of many modules—thereby achieving two goals (fixing the bug and integrating with the new interface) with only one round of recompilations.

(Note that these examples include the use of existing mechanisms that mediate interactions among programmers, such as tools for ensuring that two programmers are not simultaneously modifying the same module. In a system that explicitly supports multiple agents, it may be possible to provide this mediation from a more general perspective, and therefore to dispense with the specific tool or some particular part of the tool. Typically, the more general perspective will be preferable.)

These examples cover cases of two agents splitting the execution of one plan, dynamic decisions about how to handle the conflicts between the plans of two agents, and agents sharing information about their local plans/goals when that may affect the plans/goals of the other agents. More examples must be collected in order to characterize the nature of interactions and the support that would be useful, and to determine if existing methods of distributed AI, multiagent planning, and other formalisms for cooperation are sufficient or if new research issues are raised.

6.3.2.5 User Modeling

An important source of knowledge which we have not addressed here is knowledge about the capabilities, preferences, and habits of individual users. Incorporating this knowledge into an intelligent assistant would accomplish two goals: first, it would provide additional control over interactions with a user, and second, it would improve focusing. Knowledge about the areas in which a particular user had or lacked expertise could be used to help decide whether a possible user error should be reported to the user, or whether other explanations should be sought. It could also be used to decide how much information to communicate to a user in an error description, agenda, summary, or the like. Users are more likely to make errors when in unfamiliar territory, and more likely to need greater detail in an error description there. Knowledge about the preferences and habits of an individual user could improve focusing by directing attention to those interpretations most consistent with the patterns normally followed by that user.

6.3.2.6 User Beliefs

In the approach we have taken here, there is no distinction between what is true (or is likely to be true) in the state of the world and what the user believes to be true in the world. We have acknowledged that the planning system has less information about the world than the programmer (and provided the nonmonotonic reasoning to handle this problem), but we

have not specifically allowed for the fact that the programmer himself does not have complete information. Modeling of user beliefs may be necessary to account for certain occurrences like abandoning a plan or recognizing that a plan was executed when it should not have been. In addition, mistaken user beliefs may account for erroneous actions the user attempts to take. We give a brief example for the case of recognizing that a plan was executed when it should not have been.

Consider the knowledge that concerns the releasability of a given system version given in Figure 3.11. Recall that the previous release becomes (re-)releasable when the current release is found to be buggy. The predicate *buggy* was actually treated as a time-varying predicate—*buggy* may be false for some system version now and true later. In fact, strictly speaking, bugginess is not a time-varying predicate. Once a version is created, it either has bugs or it doesn't; what changes over time is whether or not the programmer is aware of the bugs (i.e., has found them yet.) Given this insight, there is a new way to treat the re-releasing phenomenon. When version V2 is released after V1, the programmer believes that V2 is not buggy. Later, the programmer finds that this belief was wrong, *which means that the release action was taken in error*—V2 should never have replaced V1. Therefore, it is necessary to "undo" the effects of releasing V2, which involves re-establishing V1 as the current customer release. The net effect in the state of the world *is* the same as in the Chapter 3 treatment, however representation of the actions is different (introducing an "undo" relationship between two plans).

Similarly, a programmer may start a plan based on some set of beliefs and later find that some of these beliefs are wrong. This may cause the abandonment of the specific plan in progress in favor of a new plan to achieve the same goal; or it may cause the goal to be abandoned as well. For example, the programmer may start to fix a bug, believing that fixing the bug does not depend on functionality scheduled for implementation at a later

date. Later, this is found to be false. Then, the bug fix activities are abandoned at whatever state they had progressed to, and the goal to fix the bug is deferred.

6.3.2.7 Style

The intelligent assistant as described here has no notion of what style is. By style, we mean something more substantive than preference or habit, which are factors that are somewhat probabilistic in flavor. We mean something about how problem-solving, in a plan-based context, is carried out. One way to address this issue is through the following question: given many possible plans to achieve a goal, how do these plans differ in style?

We sketch some avenues for investigation. A conservative problem-solving style is one in which those decisions that must be made first are made so as to leave the greatest number of options open for finishing the plan. The idea is to avoid ruling out options as long as possible, to retain flexibility as long as possible. An aggressive strategy would be just the opposite.

Style may also have a domain-dependent component. In the area of testing, a collection of appropriate testcases must be run, but no order is actually specified for doing this. The actual order chosen may reflect a testing style. A methodical testing style might always involve running the base cases first—a philosophy of always finding out whether backwards "progress" has been made before determining that forward progress was correctly achieved. Another testing style is to get right into testing the new function, in the hopes of getting feedback on bugs as quickly as possible (bugs are more likely in the new function than in previously implemented function).

6.3.3 AI Directions

Research directions that are primarily concerned with artificial intelligence are nonmonotonic reasoning, planning in time, and control in plan recognition.

6.3.3.1 Nonmonotonic Reasoning

There are a number of open issues concerning nonmonotonic reasoning and reconciliation as we have defined it. First, there is a need to establish the formal relationship between default logic and TMS's. A closely related issue is the formalization of reconciliation in default logic terms, building on the informal connections already described in section 4.2.2.2. We have already mentioned several open issues in reconciliation. Of particular interest is the issue of early versus late explanation discussed in section 4.1.4.1. The preference facility described in section 3.3.2 should be extended to cover other cases, such as justifications defeating each other. Finally, more work can be done in selecting among alternative solutions in reconciliation as described in section 4.3.2.1.

One of the original motivations for nonmonotonic reasoning was to overcome the limitations of existing approaches to the "frame problem"—knowing what is true in the state of the world after an action is executed. This has turned out to be extremely difficult, and is still an area of active research. Therefore, it is not surprising that the applications using nonmonotonic reasoning have dealt not with successive states of the world, but rather with successive approximations to the same state.

6.3.3.2 Planning in Time

Reasoning about successive states of the world over time, rather than successive approximations to one state in time, is one of the long-standing and still open issues in AI. Approaches that represent time explicitly include the situation calculus [McCarthy & Hayes, 1969] and formalisms based on temporal logic [Allen, 1984]. Traditional planning techniques deal with time implicitly, using a state-based model derived from the situation calculus. However, the traditional planning approach cannot accommodate simultaneous actions and has no model of past and future.

Many of the problems of reasoning about time implicitly surface in our use of nonmonotonic reasoning for deeper domain modeling. Consideration of time (i.e., changing states) affects every aspect of the nonmonotonic reasoning, including the phrasing of justifications, the locking of facts, the persistence assumptions implicit in dummy and blocked justifications, and so forth. A theory that could cover all these considerations from a general perspective would clearly be attractive.

There is another sense in which consideration of time affects planning systems, especially in intelligent assistance applications. Many tasks undertaken have deadlines (of varying degrees of formality and importance). One approach to expressing this type of information in addition to goals and using it in planning is described in [Vere, 1983]. However, additional power may be needed. For example, a problem that can arise in intelligent assistance is generating/recognizing a plan that is "good enough" given certain deadline constraints; that is, it may be better to achieve a simpler goal by a deadline than fail to complete the original goal by the deadline. The work in [Lesser et al., 1988] addresses this problem. (This is different from planning where there are constraints on the time available to generate a plan, which is a separate research topic itself; see, for example, [Dean & Boddy, 1988].)

6.3.3.3 Control in Plan Recognition

The plan recognition algorithm introduced in Chapter 2 embodies a fixed control strategy. That is, there is no attempt to reason about what to do next; it is predetermined. For example, focusing always picks a *single* "best" interpretation to pursue, even when this entails making a random choice among equally likely candidates (e.g., when the heuristics are not sufficient to discriminate completely among alternatives.) There is no representation of the extra work that might be undertaken to resolve such an uncertainty, and there is no decision-making based on the current level of uncertainty about whether to actually

undertake that work. A current research project on plan recognition that allows for reasoning about what to do next is described in [Carver, 1988].

A flexible control strategy is particularly appealing in an intelligent assistant application, where the intelligent assistant and the programmer being assisted both use the resources of a workstation. It is possible for the intelligent assistant to use extra cycles when the workstation would otherwise be idle to continue its work, even though there is not yet a new action to be recognized. (If the workstation is networked, then other idle processors on the net could be utilized as well.) This creates the opportunity both to do more computation to raise the certainty that the focus is on the right interpretation of the actions seen to date, as well as to do "advance" computation in anticipation of recognizing future actions (which could be done by invoking the planner to see what future actions are likely). This extra computation will yield better results if the decision about whether to look back or look ahead can be made dynamically—presumably, if there is a reasonable uncertainty in the current interpretation, looking ahead should be deferred in favor of looking back to rule out competing interpretations.

6.4 Summary

Achieving intelligent assistance of the type described in this dissertation is an ambitious goal. We have shown that there is existing technology—planning and plan recognition—to apply to this problem, and we have defined an approach that has the potential to achieve the goal. However, as indicated by the research in this dissertation, together with the list of open issues, the types of knowledge on which intelligent assistance can draw are many and varied. Representing and reasoning about this knowledge covers a multitude of research issues; integrating individual solutions is not merely a significant engineering problem, but also a research issue in its own right. While the idea of intelligent assistance holds much promise, there is clearly much more work to be done.

APPENDIX A

OPERATOR LIBRARY FOR SETUP-ENVIRONMENT

NOTES

These operator definitions, and the associated state schema definition that follows them, are given in the exact format required by the operator definition preprocessor. This preprocessor performs two functions: it checks the definitions for correctness and it computes the achiever relationship (discussed in section 2.2.2.1.2). Checks for correctness are at the level of syntactic form, consistency in use of variable names within an operator, etc.

A somewhat different format, geared to human—not machine—readers, has been used consistently elsewhere in this dissertation. The differences between the two formats are at the syntactic level. The most important differences are these: parameters are "declared" in an explicit list following the operator name (instead of being denoted by capitalization), top level operators are indicated with an explicit keyword, extra parentheses and punctuation are used, logical expressions are given in prefix (as opposed to infix) form, all preconditions and subgoals have names, and comments are allowed (starting with a percent sign and running to the end of the line, or between "/*" and "*/" pairs).

Conventions: File names are assumed to be unique (as they would be if always given in full path name form). The name of the directory that a file is in is treated as a separate parameter, although the information is obviously recoverable from the file name when full path names are used.

Predicate descriptions are given in comments with the state schema which appears after all the operator definitions.

```
%-------------------------------------------------------------------------------
operator(       copy(from,to,dir,contents),
%-------------------------------------------------------------------------------

                /*  This is the Unix copy command.  It copies a file.
                Parameter Descriptions:
                        from: the name of the file to be copied
                        to: the name of the file to contain the copy
                        dir: the name of the directory to contain the file containing the copy
                        contents: the contents that are being copied
                To execute this action, specify: from,to,dir
                Note: only non-destructive copies are described by this operator;
                        another operator with different preconditions and effects is needed to cover
                        destructive moves.
                */

keywords(       not_top  ),

goal(           and(    is_in(to,dir),
                        stored_in(to,contents))  ),

preconds(       from_exists(    static,   exists(file,from)),
                to_not_exists(  static,   not(exists(file,to))),
                dir_exists(     static,   exists(directory,dir))  ),

subgoals,
constraints(    stored_in(from,contents)  ),        %this constraint binds the contents parameter

effects(        new(file,to),
                add(is_in(to,dir)),
                add(stored_in(to,contents))  )  ).


%-------------------------------------------------------------------------------
operator(               move(from,to,to_dir,contents,from_dir),
%-------------------------------------------------------------------------------
                /*  This is the Unix move command.  It moves a file.
                Parameter Descriptions:
                        from: the name of the file to be moved
                        from_dir: the name of the directory currently containing the file
                        to: the name of the file after the move
                        to_dir: the name of the directory to contain the file after the move
                        contents: the contents that are being moved
                To execute this action, specify: from,to,to_dir
                Note: only non-destructive moves are described by this operator;
                        another operator with different preconditions and effects is needed to cover
                        destructive moves.
                */

keywords(       not_top  ),

goal(           and(    is_in(to,to_dir),
                        stored_in(to,contents),
```

```
                    not(stored_in(from,contents)),
                    not(is_in(from,from_dir)) ) ),

preconds(     from_exists(     static,   exists(file,from)),
              to_not_exists(   static,   not(exists(file,to))),
              to_dir_exists(   static,   exists(directory,to_dir)) ),

subgoals,
constraints(  is_in(from,from_dir),          %this constraint binds the from_dir parameter
              stored_in(from,contents) ),    %this constraint binds the contents parameter

effects(      new(file,to),
              add(is_in(to,to_dir)),
              add(stored_in(to,contents)),
              delete(stored_in(from,contents)),
              delete(is_in(from,from_dir)),
              remove(file,from) ) ).


%-------------------------------------------------------------------------------
operator(     delete(f,dir,contents),
%-------------------------------------------------------------------------------

              /*  This is the Unix delete command.  It deletes a file.
              Parameter Descriptions:
                    f: the name of the file to be deleted
                    dir: the name of the directory containing f
                    contents: the contents of f
              To execute this action, specify: f
              */

keywords(     not_top  ),

goal(         and(     not(stored_in(f,contents)),
                       not(is_in(f,dir)) ) ),

preconds(     file_exists(     static,   exists(file,f)) ),

subgoals,
constraints(  stored_in(f,contents),     %this constraint binds the contents parameter
              is_in(f,dir)    ),         %this constraint binds the dir parameter

effects(      delete(is_in(f,dir)),
              delete(stored_in(f,contents)),
              remove(file,f)  ) ).
```

```
%-------------------------------------------------------------------------------
operator(       mkdir(dir),
%-------------------------------------------------------------------------------

                /*  This is the Unix mkdir (make directory) command.
                        It creates a new directory
                Parameter Descriptions:
                        dir: the name of the new directory
                To execute this action, specify: dir
                Note: For this example, we are not concerned with the fact that directories form a tree.
                        Therefore, we have no predicate for one directory being "in" another.
                        To extend the example, this predicate would have to be added to the effects
                        of mkdir; also, the state of the world would have to include some notion of
                        the "current directory", and new directories would be placed in the current
                        directory unless a different parent_dir is specified with the mkdir command.
                */

keywords(       not_top  ),

goal(           exists(directory,dir)  ),

preconds,

subgoals,
constraints,

effects(        new(directory,dir) ) ).

%-------------------------------------------------------------------------------
operator(       define_system(sys),
%-------------------------------------------------------------------------------

                /*  This primitive operator is an abstraction covering the definition
                        of a new system version.  It is meant to represent the activities
                        of describing, formally or informally, the characteristics of the new
                        version (such as new functions to be supported, new algorithms to
                        be implemented, performance requirements, etc.)
                Parameter Descriptions:
                        sys: the "name" of the new system version
                To execute this action, specify: sys
                */

keywords(       not_top  ),

goal(           exists(system,sys)  ),

preconds,

subgoals,
constraints,

effects(        new(system,sys)  ) ).
```

```
%-------------------------------------------------------------------------------
operator(       commit_workdir(workdir,sys),
%-------------------------------------------------------------------------------

                /*  This operator captures the idea of committing a working directory
                    for use in developing a specific system version.
                Parameter Descriptions:
                        workdir: the name of the directory that will be used to hold
                                source modules being edited.
                        sys: the system version for which the workdir is committed.
                This is a complex operator, and cannot be executed directly.
                */

keywords(       not_top  ),

goal(           has_workdir(sys,workdir)  ),

preconds,

subgoals(       workdir_exists(         exists(directory,workdir)) ),

constraints(    not_used_as_workdir(workdir),   %A potential working directory cannot already be
                                                %in use as a working directory for another system
                not_used_as_refdir(workdir)  ),  %or as a reference directory for any system

effects(        add(has_workdir(sys,workdir))  )  ).

%-------------------------------------------------------------------------------
operator(       setup_workdir(workdir,sys),
%-------------------------------------------------------------------------------

                /*  This operator defines the activities needed to setup a working directory.
                    Three subtasks are defined:
                            committing the working directory (a precondition)
                            moving components into that directory (a subgoal)
                            moving extraneous file out of that directory (a subgoal)
                Parameter Descriptions:
                        workdir: the name of the directory that will be used to hold
                                source modules being edited.
                        sys: the system version for which the workdir is committed.
                This is a complex operator, and cannot be executed directly.   */

keywords(       not_top  ),

goal(           workdir_ready(sys,workdir)  ),

preconds(       workdir_committed(      dynamic,        has_workdir(sys,workdir))  ),

subgoals(       parts_in_workdir(       sys_in_dir(sys,workdir)),
                no_others_in_workdir(   only_sys_in_dir(sys,workdir)) ),
constraints;

effects         ).
```

```
%-----------------------------------------------------------------------------
operator(       commit_refdir(refdir,sys),
%-----------------------------------------------------------------------------

                /*  This operator captures the idea of committing a reference directory
                    for use in developing a specific system version. Each system
                    version is developed from an existing system version, called the
                    baseline. Components of the baseline are stored in the reference
                    directory
                Parameter Descriptions:
                    refdir: the name of the directory that will be used to hold
                        components of the baseline system.
                    sys: the system version for which the refdir is committed.
                This is a complex operator, and cannot be executed directly.
                */

keywords(       not_top  ),

goal(           has_refdir(sys,refdir)  ),

preconds,

subgoals(       refdir_exists(          exists(directory,refdir))  ),

constraints(    not_used_as_workdir(refdir)  ),

effects(        add(has_refdir(sys,refdir))  ).

%-----------------------------------------------------------------------------
operator(       setup_refdir(refdir,sys,baseline),
%-----------------------------------------------------------------------------

                /*  This operator defines the activities needed to setup a reference directory.
                    Three subtasks are defined:
                        committing the reference directory (a precondition)
                        moving components into that directory (a subgoal)
                        moving extraneous file out of that directory (a subgoal)
                Parameter Descriptions:
                    refdir: the name of the directory that will be used to hold
                        the baseline system version.
                    sys: the system version for which the refdir is committed.
                    baseline:  the existing system version from which sys will be
                        developed.
                This is a complex operator, and cannot be executed directly.
                */

keywords(       not_top  ),

goal(           refdir_ready(sys,refdir)  ),

preconds(       baseline_constructed(   static,         constructed(baseline)),
                                        %We assume that an existing system version is not
                                        %a candidate baseline unless work on it has finished
```

```
                refdir_committed(       dynamic,        has_refdir(sys,refdir)) ),

subgoals(       baseline_in_refdir(     sys_in_dir(baseline,refdir)),
                no_others_in_refdir(    only_sys_in_dir(baseline,refdir)) ),

constraints(    refdir_consistency(baseline,refdir,sys) ),      %Two system versions can share the
                                                                %same reference directory as long as
                                                                %they are using the same baseline

effects(        add(baseline(sys,baseline)) ) ).    %As a by-product of setting up the reference
                                                    %directory, we learn which baseline in is use
                                                    %and so record this in the world state.

%-------------------------------------------------------------------------------
operator(       remove_extra_files(dir,sys,somefile,contents),
%-------------------------------------------------------------------------------

                /*  This operator defines the activities needed to remove files from a specified directory;
                        the files being removed are all those files in the directory that do not
                        contain part of a specified system version.
                        There is one subtask that is iterated:
                                moving an extraneous file out of the directory (a subgoal)
                    Parameter Descriptions:
                        dir:  the directory from which extra files are to be removed
                        sys: the system version whose parts are allowed in dir.
                        somefile: a file to be moved out of dir
                        contents:  the contents of somefile
                    This is a complex operator, and cannot be executed directly.
                    */

keywords(       not_top ),

goal(           only_sys_in_dir(sys,dir) ),

preconds,

subgoals(       moved_out_of_dir(       not(is_in(somefile,dir)),
                                        iterated_until(only_sys_in_dir(sys,dir))) ),

constraints(    stored_in(somefile,contents),       %These constraints ensure that only the
                not_in_sys(sys,contents) ),         %right files are allowed to remain in dir

effects                 ).
```

```
%-------------------------------------------------------------------------------
operator(       place_parts(dir,sys,somefile,contents),
%-------------------------------------------------------------------------------

                /*  This operator defines the activities needed to place files in a specified directory;
                        the files being placed are all those files that contain part of a specified
                        system version.
                        There is one subtask that is iterated:
                                moving a file into the directory (a subgoal)
                Parameter Descriptions:
                        dir:  the directory into which files are placed
                        sys: the system version whose parts are allowed in dir.
                        somefile: a file to be moved into dir
                        contents:  the contents of somefile
                This is a complex operator, and cannot be executed directly.
                */

keywords(       not_top ),

goal(           sys_in_dir(sys,dir) ),

preconds,

subgoals(       moved_to_dir(   and(    is_in(somefile,dir),
                                        stored_in(somefile,contents)  ),
                                iterated_until(sys_in_dir(sys,dir)) ) ),

constraints(    part_of(sys,contents) ),    %The contents of the file being moved into dir
                                            %must be part of system version sys.

effects         ).

%-------------------------------------------------------------------------------
operator(       setup_env(system,refdir,workdir),
%-------------------------------------------------------------------------------

                /*  This operator defines the activities needed to setup an environment for the
                        development of a new system version
                        There are four subtasks:
                                defining  the new system version (a precondition)
                                setting up the reference directory (a subgoal)
                                setting up the working directory (a subgoal)
                                creating the new parts for the new system version (a subgoal)
                Parameter Descriptions:
                        system: the system version to be developed.
                        refdir: the reference directory for system
                        workdir:  the working directory for system
                This is a complex operator, and cannot be executed directly.
                */

keywords(       not_top  ),

goal(           env_setup(system,refdir,workdir)  ),
```

```
preconds(       system_defined( dynamic,        exists(system,system)) ),

subgoals(       refdir_ready(   refdir_ready(system,refdir)),
                workdir_ready(  workdir_ready(system,workdir)),
                parts_ready(    parts_ready(system)) ),

constraints,

effects         ).

%-------------------------------------------------------------------------------
operator(       construct(system,refdir,workdir,load_mod,file),
%-------------------------------------------------------------------------------

                /*  This operator is an abstraction that covers compilation and linking
                    of a system version. (Actually, the creation of object modules
                    via compilation is not shown in detail.) Construct is presented
                    here as a primitive operator, but in an extended example it would
                    be complex with separate subgoals for these two activities.
                Parameter Descriptions:
                    system: the system version being developed
                    refdir: the reference directory used by system
                    workdir:  the working directory used by system
                    load_mod:  the load module (executable incarnation) of system
                    f:  the file containing the load module of system
                To execute this action, specify: system
                */

keywords(       toplevel ),

goal(           constructed(system) ),

preconds(       env_setup(      dynamic,        env_setup(system,refdir,workdir) ) ),

subgoals,
constraints(    not_equal(directory,refdir,workdir) ),

effects(        new(contents,load_mod),                 %linking creates a new load module
                new(file,f),                            %a new file is also created
                add(stored_in(f,load_mod)),             %the load module is stored in the new file
                add(is_in(f,workdir)),                  %the new file is in the working directory
                add(kind(load_mod,load)),               %the contents of the new file is of type
                                                        %"load" (as opposed to "source")
                add(executable(system,load_mod)),       %the new load module is the executable
                                                        %for system
                add(constructed(system)),
                delete(has_refdir(system,refdir)),      %the reference directory is no longer
                                                        %committed for system
                delete(has_workdir(system,workdir)) ).  %the working directory is no longer
                                                        %committed for system
```

```
%-------------------------------------------------------------------------------
operator(       make_new_parts(sys,comp),
%-------------------------------------------------------------------------------

                /*  This operator defines the activities of making the new source modules
                        (components) that are needed to implement a new system version.
                        There is one subtask that is iterated:
                        creating a new component (a subgoal)
                        A new system version is required to have at least one new component,
                        but the exact number of components is otherwise not constrained.
                Parameter Descriptions:
                        sys: the system version which is being constructed.
                        comp:  a new component (source module version) for sys
                This is a complex operator, and cannot be executed directly.
                */

keywords(       not_top  ),

goal(           parts_ready(sys) ),

preconds,

subgoals(       part_created(       part_of(sys,comp),
                                    iterated(1)  ) ),

constraints,

effects(        add(parts_ready(sys))       )       ).

%-------------------------------------------------------------------------------
operator(       create(sys,somefile,comp_unit,dir),
%-------------------------------------------------------------------------------

                /*  This operator represents an abstraction for editing a new source module
                        version to make a component for a new system version being constructed.
                        All the detail of how many times the editor is actually invoked to
                        make changes to this source module is hidden; to show this detail
                        create would be made into a complex operator with an iterated subgoal
                        that was achieved by individual editor invocations.  For this version of
                        create, we assume that the new source module is "created" in a new file that
                        is contained in a specified existing directory.
                Parameter Descriptions:
                        sys: the system version which is being constructed.
                        comp_unit:  a new compilation unit for sys
                        somefile:  the file containing comp_unit
                        dir:  the directory containing somefile
                To execute this action, specify: somefile,comp_unit,dir
                */

keywords(       not_top  ),

goal(           part_of(sys,comp_unit) ),
```

```
preconds(       no_such_file(    static,   not(exists(file,somefile))),
                dir_exists(      static,   exists(directory,dir))  ),

subgoals,

constraints,

effects(        new(contents,comp_unit),
                new(file,somefile),
                add(kind(comp_unit,source)),
                add(stored_in(somefile,comp_unit)),
                add(is_in(somefile,dir)),
                add(part_of(sys,comp_unit))        )        ).

%-------------------------------------------------------------------------------
%       State Schema
%-------------------------------------------------------------------------------

/*
```

The following statement declares all the entity types that are used in the state schema. A distinction is made between *files* and their (current) *contents*. Files are contained in *directories*. *Systems* correspond to conceptual entities rather than physical entities in the computer.

```
*/

entities(       [file,directory,contents,system]  ).

/*Each predicate used in the state schema must be defined. The definition establishes the predicate name,
the number of arguments, and the entity types of those arguments. For example, stored_in is a predicate
with two arguments, the first of which is a file and the second of which is a contents. This information is
used to check predicate usage within operator definitions so that trivial errors in writing operators (such as
reversing the order of arguments) can be detected when the operator library is first processed. A predicate
argument can be a literal; see for example, the definition of kind (whose intended literal values are source
and load.)

Predicates also have attributes. The first attribute describes the type of the predicate: core or computed. A
core predicate can appear in an effects clause, while a computed predicate cannot (but must have a macro
definition by which it can be computed). When the deeper domain modelling techniques of Chapters 3-4 are
used, then predicates may also be denoted extended, in which case nonmonotonic rules are used to determine
their truth/falsity. The second predicate attribute describes the behavior of the predicate over time: some
predicates vary in truth value while others are fixed. This information is used only with the deeper domain
modelling (it helps to instantiate justifications and to lock predicates.)
*/

predicate_def(      stored_in(file,contents),                    core,    varies).
                        %a distinction is made between files—places to store things,
                        %and contents—things stored in places,
                        %this predicates records the current correspondences
predicate_def(      is_in(file,directory),                       core,    varies).
                        %files are contained in directories, as in UNIX
predicate_def(      has_refdir(system,directory),                core,    varies).
                        %directories can be committed as reference directories for systems
predicate_def(      has_workdir(system,directory),               core,    varies).
```

```
                    %directories can be committed as working directories for systems
predicate_def(      kind(contents,literal),                 core,   fixed).
                    %contents can be source modules or load modules or other
predicate_def(      executable(system,contents),            core,   fixed).
                    %the load module of a system is its executable
predicate_def(      baseline(system,system),                core,   varies).
                    %one system can be the baseline from which another is developed
predicate_def(      part_of(system,contents),               core,   varies).
                    %the parts of a system are its source modules
predicate_def(      parts_ready(system),                    core,   varies).
                    %when editing is done, the parts of a system are ready
predicate_def(      constructed(system),                    core,   varies).
                    %a system that has had a load module built is constructed.

predicate_def(      refdir_ready(system,directory),         computed,varies).
predicate_def(      workdir_ready(system,directory),        computed,varies).
predicate_def(      sys_in_dir(system,directory),           computed,varies).
predicate_def(      represented_in_dir(contents,directory), computed,varies).
predicate_def(      only_sys_in_dir(system,directory),      computed,varies).
predicate_def(      refdir_consistency(system,directory,system), computed,varies).
predicate_def(      not_used_as_refdir(directory),          computed,varies).
predicate_def(      not_used_as_workdir(directory),         computed,varies).
predicate_def(      not_in_sys(system,contents),            computed,varies).
predicate_def(      env_setup(system,directory,directory),  computed,varies).

/*

Each computed predicate requires a macro by which its truth value is determined. These are given below.
Each macro definition has three parts. First, we see the computed predicate and its visible arguments. Then
there is a list of all the arguments that are used in the macro. Then, the macro is given. For example, the
computed predicate represented_in_dir has two visible arguments named stuff and dir, but three arguments
altogether: stuff, dir and somefile. The macro for represented_in_dir is a simple conjunction of two other
predicates: stored_in and is_in, both of which are core predicates. Some computed predicates use other
computed predicates in their macro definitions, which is fine as long as there is no circularity involved.
*/

computed_pred(      represented_in_dir(stuff,dir), [stuff,somefile,dir],
                        and(    stored_in(somefile,stuff),
                                is_in(somefile,dir)) ).
                    %Something is said to be represented in a directory if there is
                    %a file that contains it and that file is in that directory.

computed_pred(      refdir_ready(sys,dir), [sys,dir,base],
                        and(    baseline(sys,base),
                                exists(directory,dir),
                                sys_in_dir(base,dir),
                                only_sys_in_dir(base,dir)) ).
                    %A reference directory is ready for a system if the baseline
                    %of the system is in the directory and only the baseline is
                    %in the directory
```

```
computed_pred(      workdir_ready(sys,dir), [sys,dir],
                        and(    exists(directory,dir),
                                sys_in_dir(sys,dir),
                                only_sys_in_dir(sys,dir))  ).
                        %A working directory is ready for a system if  the
                        %system is in the directory and only that system is in the directory

computed_pred(      sys_in_dir(sys,dir), [sys,dir,stuff],
                        and(    exists(system,sys),
                                exists(directory,dir),
                                not(    and(    part_of(sys,stuff),
                                                not(represented_in_dir(stuff,dir))))) ).
                        %A system is in a directory if each of its parts is contained in a file
                        %that is in that directory.

computed_pred(      only_sys_in_dir(sys,dir), [sys,dir,stuff,file],
                        and(    exists(system,sys),
                                exists(directory,dir),
                                not(    and(    is_in(file,dir),
                                                stored_in(file,stuff),
                                                not(part_of(sys,stuff)) ) ))  ).
                        %Only system sys is in a directory if every file in the directory
                        %contains a part of sys.

computed_pred(      refdir_consistency(base,dir), [base,dir,anysys],
                        and(    exists(directory,dir),
                                exists(system,base),
                                exists(system,sys),
                                not(    and(    has_refdir(anysys,dir),
                                                not_equal(system,anysys,sys),
                                                not( baseline(anysys,base))))  )  ).
                        %Use of a reference directory to contain a baseline system version base
                        %is consistent if all other systems to which the directory is a reference
                        %directory also use base as their baseline.

computed_pred(      not_in_sys(sys,cont), [s,cont,sys],
                        and(    exists(system,sys),
                                exists(contents,cont),
                                not(  and(      part_of(s,cont),
                                                equal(system,s,sys))) )  ).
                        %A contents cont is not in a system sys  if it is part of another system
                        %or part of no system at all.

computed_pred(      not_used_as_refdir(dir), [dir,anysys],
                        or(     and(    exists(system,sys),
                                        not(has_refdir(anysys,dir))  ),
                                not(    exists(directory,dir))  )  ).
                        %A directory is not in use as a reference directory if there is no
                        %system to which it is committed as a reference directory.
```

```
computed_pred(      not_used_as_workdir(dir), [dir,anysys],
                        or(     and(    exists(system,sys),
                                        not(has_workdir(anysys,dir))  ),
                                not(    exists(directory,dir))  )  ).
                    %A directory is not in use as a working directory if there is no
                    %system to which it is committed as a working directory.
```

APPENDIX B

ALGORITHM FOR PLAN RECOGNITION

Given: one interpretation (plan network and corresponding world state) in the interpretation tree that is the current "focus" of attention. Assume this interpretation covers actions 1 through N-1.

START: Get action N.

Generate candidate interpretations for N:

Finding all possible paths to account for action N uses the pre-computed reachability information. This information is stored in the form:

reachable (op1,part1,op2,op3, part3),

where op1, op2 and op3 are operators, part1 is a precondition or subgoal of op1, and part3 is a precondition or subgoal of op3. This means that there is a path from op1/part1 to primitive operator op2 and a possible first step on the path (downward from op1 to op2) is the step to op3/part3. If there are no intervening steps between op1/part1 and op2, then op3/part3 are omitted.

Initialize the set PATHS of paths as follows. (Each step on a path will be represented by an operator paired with a specific precondition or subgoal of that operator. Paths start at top level operators and step "downwards", ending eventually at a primitive operator.)

Consider all operator instances O in the current interpretation and all their instantiated (e.g., pending) preconditions P or subgoals S. For each combination of O/P and/or O/S that is reachable from action N, the existing path from the top level operator TOP to O/P or O/S is an element of S. These elements of PATHS are

extension paths. The dynamic namescopes of these paths are the dynamic namescopes of their respective TOPs.

Now consider all top level operators, regardless of whether instantiated. For each top level operator O and each of its preconditions P or subgoals S that are reachable from action N, make a (one-step) path consisting of O/P or O/S in PATHS. These elements of PATHS start new plans. Initialize a new dynamic namescope for each of these paths, assigning new dynamic names to the parameters of O.

Now extend the paths in PATHS all the way to action N. Remove from PATHS each path P that does not end with a primitive operator (equal to action N), and do the following:

Let O1/PS1 be the last step on path P. For each O3/PS3 such that:
reachable(O1, PS1, Action N, O3, PS3),
add a path to PATHS that consists of P extended to include O3/PS3 as the last step.

Retrieve the static-to-static namemap associated with using O3 to achieve PS1 of O1 (if there are two or more possible namemaps, make as many copies of the path as there are namemaps and repeat the name calculations using a different namemap on each path). Use this namemap, and the static-to-dynamic namemap of O1 to compute the static-to-dynamic namemap of O3, assigning new dynamic names to any parameters of O3 that are not otherwise mapped.

Prune candidates:

Check each path P in PATHS as follows:

If P extends an existing plan, record the bindings of action N under the appropriate dynamic names for the plan and reject P if binding inconsistencies are detected.

If P starts a new plan, then initialize the appropriate dynamic names for that plan with the bindings of action N.

Traverse P from action N to the top, considering each operator O on the path:

Check preconditions of O:

If path P reaches O through a precondition of O, or if O has already started, do nothing. Otherwise, check that all the preconditions of O are satisfied. If any precondition check fails, make a note of that fact.

Check constraints of O:

Omit constraint checks entirely if O has already started. Omit consideration of a constraint if path P reaches O through a precondition of O and some parameters in the constraint are not

bound. Test all other constraints, rejecting P if any tested constraint is false.

Check goal of O:

If there are bindings for all parameters in the goal of O, then check to see that the goal is not already true. If this check fails, reject P.

Check to see that the goal of the top level operator in P does not duplicate the goal of the top level operator in any other existing plan. Omit this check if there are any unbound parameters in the goal at the top of P. Reject P if the check is made and fails.

Instantiate a new interpretation that incorporates P (unless P has already been rejected). Let I be the interpretation (at level N-1) that is the current focus; I will be the parent of the new interpretation. Perform the instantiation by duplicating I (both plan network and world state), then adding to I the new part of the path P and new bindings (taken from action N or discovered as part of processing P).

Focus among interpretations at level N:

If there are no interpretations at level N (or if all such interpretations have been rejected), inform user of possible error in action N. User may choose to perform action N anyway (causing recognizer to work harder to find an interpretation for action N) or to perform a different action. Depending on user response, either:

Reject parent interpretation at level N-1, and then backtrack (chronologically) as follows:

Re-focus by tracing back through interpretation tree to nearest ancestor with at least one remaining child interpretation (call the level of the child level J). Select from among competing children if necessary, and update status of selected interpretation SI (see algorithm below).

Start cycle over on action J+1 with new focus SI, continuing cycles until all actions including N are accounted for.

Or, terminate cycle (delete all interpretations at level N, then return to START and wait for new action N).

Otherwise, select among interpretations with heuristics. Replace current focus (at level N-1) with selected focus (at level N). (The heuristics imply that, if possible, the selected interpretation will involve a path P that extends an existing plan and will not require planning to satisfy any preconditions.)

Update status of new focus (at level N):

If the path P on which this interpretation was constructed involved any failed preconditions, call planner to achieve them all. If planning fails, reject this interpretation; repeat focusing at level N, as described above.

Remove from protection list the preconditions of all operators that "started" with this action (e.g., all operators whose preconditions were actually tested during processing of path P).

Traverse the path P from action N to the top:

At each operator O:
Check for completion (e.g., goal of O true).
If completed, post effects of operator.

At each precondition or subgoal:
Check for satisfaction (e.g., precondition or subgoal true).
If satisfied, put on protection list.

Halt traversal at first incomplete operator, or unsatisfied precondition or subgoal.

Process pending conditions:

Check each pending (dynamic) precondition:
If now true, put precondition on protection list.
If all dynamic preconditions of operator are now true, instantiate the subgoals of operator.

Check each pending subgoal:
If now true, put subgoal on protection list.
If all subgoals of operator are now true, post effects of operator and take subgoals off protection list.

Repeat processing of pending conditions until no further effects are posted.

Check protection list for violations:

Find all conditions (preconditions or subgoals) on protection list that are now false. If there are no violations, then updating is complete.

Otherwise, inform user of violations. User has three options: to perform action in spite of the violations, to take a different action, or to force the recognizer to find another interpretation of the action (in which presumably there are no protection violations.)

Depending on user response, either:

Take conditions off protection list, and mark them unsatisfied.

Or, terminate cycle, deleting all interpretations at level N, changing focus back to parent interpretation at level N-1, and returning to START to wait for new action N.

Or, reject this interpretation. If no remaining interpretations at level N, backtrack; otherwise, focus at level N again.

APPENDIX C
PLAN RECOGNITION EXAMPLES

All figures in this appendix appear at the end of the appendix, starting on page 287.

Simple Recognition Example

The first recognition example shows the basics of plan recognition: how the search proceeds, and how interpretations are identified, pruned, and updated.

The example consists of an initial world and a sequence of actions. The initial world state in which the first action is performed has one existing system, *s1*, which has been constructed; *s1* has one part (e.g., one source module), *s1_unit*, stored in file *unit_file*. There is only one directory, *d1*, and *unit_file* is in *d1*. Also stored in *d1* is another file, *wildfile*, whose contents are *foo*. The action sequence is as follows:

1. define_system(system=s2)
2. mkdir(dir=d2)
3. move(from=unit_file,to=base_file,to_dir=d2)
4. mkdir(dir=d3)
5. create(somefile= new_file,comp_unit=s2_unit,dir=d3)
6. construct(system=s2)

A preview of the actual search path taken through the interpretation tree is shown in Figure C.1; six trees are shown, each one corresponding to the search progress after the next action in sequence is recognized (the Nth tree shows the search after recognizing the first N actions). Interpretation 1 (shown in Figure C.2) represents the initial conditions: a world state as described above and an empty plan network (no on-going activities in progress). As indicated in Figure C.1, interpretation 3 is rejected because it cannot be extended to incorporate an explanation for action 3. Its sibling, interpretation 4, becomes the new focus, and the new context in which recognition of action 3 is attempted (and

succeeds). Interpretation 6 is never fully updated, and its progeny are never explored, because recognition is successful in a descendent of its competitor, interpretation 5.

In the following paragraphs, we describe how the interpretations are generated for each new action. Each major section, corresponding to one "cycle" of the plan recognition algorithm (see algorithm in Appendix B), covers recognition of one action in the context of the current focus.

1. Action 1 (Define-system) in Interpretation 1: The initial focus is on interpretation 1 (Figure C.2). There is only one possible path from *define_system* to a top-level operator (as can be seen in Figure 2.6). Since the plan network net is initially empty, there is no way to continue any plan that is in progress; the only alternative is to start a new plan This path from *define-system* to *construct* is shown in Figure C.3; it meets all the semantic validity tests (preconditions and constraints satisfied, etc.) and is instantiated as interpretation 2. The status of the plan network and world state are updated, by marking *define-system* as completed, performing its effects, marking *system-defined* (the precondition to *setup-env*) as satisfied, and instantiating the subgoals of *setup-env* (because all its dynamic preconditions are satisfied). Interpretation 2, fully updated, is shown in Figure C.4.

2. Action 2 (Mkdir) in Interpretation 2: There are two paths from *mkdir* to a top-level operator (as seen in Figure 2.6). Thus, there are four possible ways to add a rationale for *mkdir* to interpretation 2: two ways to extend the existing plan (that started with *define-system*) and two ways to start a new plan concurrent with the existing plan.

The two paths from *mkdir* to the top-level, one through *setup-refdir* and one through *setup-workdir*, both connect to the existing plan in interpretation 2, one through the pending subgoal for *refdir-setup* and one through the pending subgoal for *workdir-setup*. Both of these paths meet all the semantic constraints, and so are instantiated as interpretations 3 and 4.

In examining the semantics of the two paths from *mkdir* to a new top-level plan, we find three conflicting requirements. *Exists(system)* is a precondition to *setup-env*; it is only true for s1 and s2. When we then apply the goal-not-satisfied test on *construct* (the new top-level plan), we learn that the system cannot be s1—it is already constructed. But now, the goal of the new top-level plan is *constructed(s2)* which duplicates the goal of the only top-level plan already in the plan network. So, there is no way that this action could be starting a new top-level plan.

Of four syntactically legal possibilities, two have been pruned so that only two are actually instantiated. Note that the choice of focus between interpretations 3 and 4 is arbitrary (neither the first focusing heuristic, to

prefer continuing an old plan to starting a new one, nor the second one, to prefer interpretations that do not require planning, discriminate between these two cases). The (arbitrary) decision to focus on interpretation 3 just happens to be the *incorrect* decision, although that cannot be determined with the information available at this point in time. Interpretation 3, fully updated, is shown in Figure C.5. Note that *workdir-setup* has been fully satisfied; this is correct, if surprising. Since *s2* has no parts, all parts of *s2* are in *d2*; since *d2* is empty, no extraneous files are there. Thus the two subgoals in *setup-workdir* are satisfied, and *setup-workdir* is regarded as complete.

3. Action 3 (Move) in Interpretation 3: There are four paths from *move* to the top (refer to Figure 2.6). That gives eight possibilities, four ways to continue the single existing plan and four ways to start a new plan.

First consider the ways that these four paths might be involved in extending the only plan we have in progress in interpretation 3. The first path, through *place-parts* to *setup-refdir*, satisfies the pending subgoal *refdir-setup* in the plan network of interpretation 3, and binds the reference directory parameter to *d2*, the target directory in this move. But in interpretation 3, *d2* is the working directory (because of the role played by the previous action, *mkdir*). Thus, the constraint in *construct* that the working and reference directories not be the same is violated, and the path fails the semantic tests. The second path, through *remove-extra-files* to *setup-refdir*, also fails. The file being moved out of *d1* is *unit_file*, which contains a part of *s1*. Therefore, *s1* cannot be the baseline from which *s2* will be constructed; but that leaves no system that is currently constructed to be the baseline of *s2*, and so the precondition on *setup-refdir* fails.

The two paths involving *place-parts* to *workdir-setup* and *remove-extra-files* to *workdir-setup* are not syntactically viable because *workdir-setup* is not a pending subgoal—it has already been satisfied via the *mkdir*. Therefore these paths cannot continue the current plan.

When paths starting a new top-level plan are considered, three possibilities fail for a common reason: duplication of top-level goal. (The reasoning is the same as that for new paths in action 2.) In the fourth case (*move* to *place-parts* to *setup-workdir*), the binding of parameters is such that the top-level goal must be *constructed(s1)*; but this goal is already true, so again the path is ruled out.

Of the eight possibilities considered, all were rejected. Thus, interpretation 3 is inconsistent with action 3. Using the simple strategy of assuming that the focus is always right (see section 2.2.4.3), the plan recognizer now gives the user the choice of selecting another action or going ahead with this one.

Assuming that the user chooses to go ahead with this action, interpretation 3 is rejected and the focus shifts to its next most likely (actually only remaining sibling), interpretation 4. Interpretation 4 is now updated (this work was postponed when the interpretation was originally instantiated) and is shown in updated form in Figure C.6.

4. Action 3 (Move) in Interpretation 4: The objective now is to recognize the same action (the *move* which has not yet been explained) in a new context—interpretation 4. As before, there are four paths from *move* to the top, and eight possibilities to consider.

The first path, through *place-parts* to *setup-refdir*, satisfies the pending subgoal *refdir-setup* in the plan network of interpretation 4; this path is consistent with the fact that we have already determined the reference directory to be *d2*. This path becomes interpretation 5. The second path, through *remove-extra-files* to *setup-workdir*, satisfies the pending subgoal *workdir-setup* in the plan network of interpretation 4, and binds the working directory parameter to *d1*. (The legality of this path depends on planning to satisfy *has-workdir(s2,d1)*, the precondition of *setup-workdir*; the details of this are deferred to a later example). This path becomes interpretation 6.

The third path, through *remove-extra-files* to *setup-refdir*, fails on constraints: only moves out of *d2* are appropriate, given that *d2* is known to be the reference directory. The fourth path, through *place-parts* to *setup-workdir*, also fails on constraints: the contents of the file being moved are not part of *s2*.

When paths to a new top-level plan are considered, all four possibilities fail, exactly as in the previous cycle.

Of the eight syntactically legal possibilities, only two survive semantic tests to be instantiated. The decision to focus on interpretation 5 is made on the basis that interpretation 6 requires planning, whereas interpretation 5 does not. This choice, which is potentially fallible, will in fact lead to successful interpretation of all actions. Interpretation 5 is shown in Figure C.7.

5. Action 4 (Mkdir) in Interpretation 5: There are two paths from *mkdir* to the top-level, one through *commit-workdir* and one through *commit-refdir*. *Commit-workdir* satisfies a pending subgoal (*workdir-committed*) in the plan network of interpretation 5, and this path becomes interpretation 7. The subgoal *refdir-committed* is already satisfied, so there is no way that the path through *commit-refdir* can continue the existing plan.

Considering both paths as ways to start a new top-level plan again fails on the duplication of top-level goal check.

Of the four possibilities, only one survives all checks to be instantiated. It automatically becomes the selected focus. Interpretation 7 is shown in Figure C.8.

6. **Action 5 (Create) in Interpretation 7**: There is only one path from *create* to a top-level operator. This path could contribute to the pending subgoal *parts-created* in the plan network of interpretation 7; since it passes the semantic checks, it is instantiated as interpretation 8. The possibility that this path starts a new top-level plan fails the top-level goal duplication check.

Of the two syntactically legal cases, only one is actually instantiated, and it automatically becomes the chosen focus. Interpretation 8 is shown in Figure C.9.

7. **Action 6 (Construct) in Interpretation 8**: *Construct* is a top level operator. One possibility is that this instance of *construct* is the completion of the plan already in the plan network; this becomes interpretation 9. (To make this path meet the semantic checks, it is necessary to do planning: the iterated subgoal in *create-parts* must be assumed to have completed, which will complete the remaining unsatisfied subgoal of *setup-env*, thereby satisfying the precondition to *construct*. The third example treats this planning in detail.) The possibility that *construct* starts a new top-level plan is ruled out by the top-level-goal-duplication test.

Of two possibilities, only one is instantiated. This interpretation, when fully updated, shows a plan network containing one plan which has now been performed in entirety. It is shown in Figure C.10.

Protection Violation Example

The second example shows a protection violation situation. The initial world state and first four actions are as in the first example. The fifth action is different (the operation is the same but the parameters are not) and is:

5. create (somefile=new_file,comp_unit=s2_unit,dir=d2)

In the first recognition example, action 5 is recognized in the context of interpretation 7, which is reproduced in Figure C.11. In this interpretation, there are two protected subgoals (now highlighted in C.11). The operator *setup-env* is in progress, and two of its three subgoals have been satisfied. Each subgoal was marked protected when the operator achieving it completed successfully; since *setup-env* has not yet completed, these subgoals are still on the protected list.

The proposed action (which differs from action 5 of the first example in that the creation is done in directory d2 rather than d3) violates both these protected subgoals. It violates the subgoal *refdir-ready* because it results in an extraneous file being placed in the

reference directory, which previously contained all parts of system *s1* and only those parts. It violates the subgoal *workdir-ready* because it results in a part of system *s2* being stored in a directory other than *d3*. (Note that when *s2* had no parts and directory *d3* was empty, *workdir-ready* was true—all parts of *s2* were in *d3* and no others were in *d3*.)

The violation is detected during the updating of the plan network and world state to reflect the effects of action 5. That action has an interpretation (to help satisfy the third subgoal of *setup-env*) that is legal in all respects other than the protection violation. The user is given the choice of going ahead with this action anyway (thereby accepting these violations), choosing another action to replace this action, or forcing the recognizer to work harder to recognize action 5. If the user chooses the second option, recognition of the replacement action would be attempted in the context of interpretation 7 (Figure C.11). In that case, recognition proceeds as if the erroneous action had never been seen (and of course the erroneous action has been prevented from executing).

If the violations are accepted, then the two subgoals that are violated need to be marked not-satisfied and taken off the protected list. This is done in addition to all the normal changes that result from recognizing an action. The interpretation that results from going ahead with action 5 are shown in Figure C.12. Note that the two operators *setup-refdir* and *setup-workdir* are no longer shown as *achievers* of the subgoals *refdir-ready* and *workdir-ready* respectively, but rather as *contributors* towards these subgoals; the fact that *d2* and *d3* are committed as reference and working directories respectively has not been changed in the world state. The formerly protected subgoals are now awaiting new achievers.

In the context of Figure C.12, a single action (moving *new_file* from *d2* into *d3*) would reachieve both these subgoals simultaneously. Such a *move* would have two possible interpretations: *move* to *remove-extra-files* to *setup-refdir* or *move* to *place-parts* to *setup-workdir*. No matter which interpretation is chosen, one subgoal—*refdir-ready* or *workdir-ready*—will be satisfied directly and the other subgoal—*workdir-ready* or *refdir-ready*—will be satisfied as a side-effect. (The recognition algorithm will pick one interpretation arbitrarily; the interpretation will correctly reflect the state of the plans, but the fact that a competing interpretation is essentially identical will be ignored, as described in section 2.2.4.2. That means that it is possible for the competing interpretation to be visited during backtracking; this is not desirable, since it is a waste of time. However, detecting that the two interpretations are essentially the same requires recognizing that, under certain bindings, two apparently different subgoals in the plan network are the same; doing this detection also takes time. The trade-off decision is not trivial.)

Planning in Plan Recognition Example

The third, and final, example shows how planning is carried out during plan recognition. The initial world state in which action 1 is performed has one existing system, *s1*, which has been constructed; it has one part, *s1_unit*, stored in file *unit_file*. There are two directories, *d1* and *d2*. *Unit_file* is in *d1*, and *d2* is empty. The series of actions are:

1. define_system(system=s2)
2. create(somefile=unit_file,comp_unit=s2_unit, dir=d2)
3. construct(system=s2)

Actions 1 and 2 each have a single interpretation; if we call the initial situation interpretation 1 (see Figure C.13), then these are interpretations 2 and 3. They are shown in Figures C.14 and C.15 respectively. Interpretation 3 (Figure C.15) is the context in which action 3 must be recognized. The legality of action 3 depends on its precondition, *env-setup*, being true, but this precondition is not satisfied in interpretation 3. Therefore planning is invoked. The effects of planning on the plan network are shown in the dotted area in Figure C.16. The results of planning give a revised version of interpretation 3, in which no new explicit actions are involved, but in which the state of the world and the state of the plan network have been changed.

The following is a step-by-step account of how the planner proceeds. In interpretation 3 (Figure C.15), the precondition *env-setup* appears in a state of partial completion through the plan *setup-env*. The precondition of *setup-env* has already been satisfied, but the three subgoals are not yet satisfied; we must plan to satisfy each one.

To satisfy the subgoal *refdir-ready*, planning finds and instantiates the complex plan *setup-refidr*. The two subgoals of this operator are satisfied if we take *s1* to be the baseline and *d1* to be the reference directory. However, the precondition, *refdir-committed*, is not satisfied. In order to satisfy it, planning finds and instantiates the complex operator *commit-refdir*. Its only subgoal is satisfied for the directory *d1*, so we can complete *commit-refdir*, and post its effects (recording that *has-refdir(s2,d1)* is true). With this, *setup-refdir* is now completed, so we can post its effects (recording that *baseline(s2,s1)* is true). Satisfaction of the second subgoal of *setup-env* proceeds in a very similar fashion.

The third and final subgoal of *setup-env* is already in progress via the operator *create-parts*. Its single subgoal, which is *iterated(1)*, has been satisfied once via the *create* operator. Since the minimal number of required iterations has been achieved, we can regard the subgoal as having completed. This completes *create-parts*, whose effects can now be posted, adding *parts-ready(s2)* to the world state. With this, *setup-env* completes, and its effects can be posted (actually there are none). Now, the precondition to *construct* is satisfied, so planning is both complete and successful.

The changes made during planning, all shown in Figure C.16, include instantiation of new operators (*setup-refdir*, *setup-workdir*, *commit-refdir*, *commit-workdir*), completion of these operators, completion of existing operators (*create-parts*, *setup-env*), and various changes (actually all additions) to the world state. Figure C.16 represents interpretation 3', a revision to interpretation 3, not a new interpretation The recognition of the final action (*construct*) is now possible in the context of interpretation 3'.

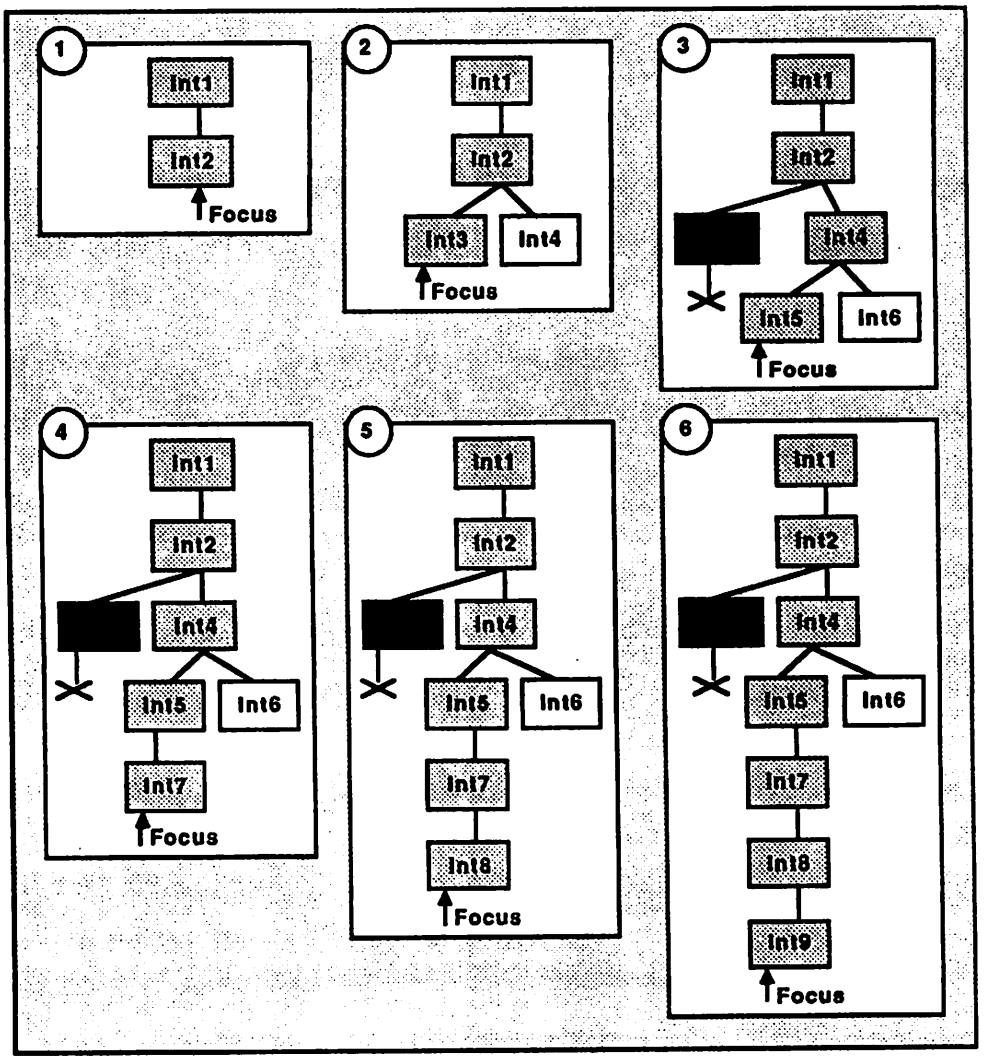

Figure C.1 First Example: Overview of Search

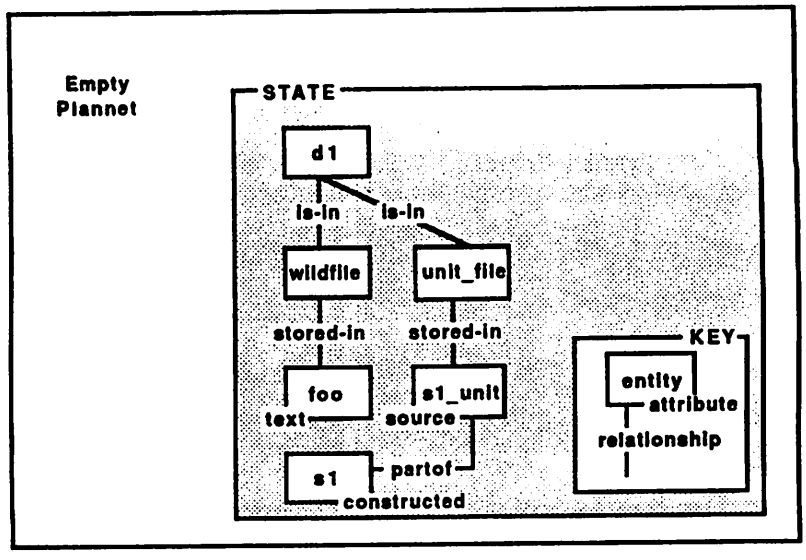

Figure C.2 First Example: Initial Interpretation

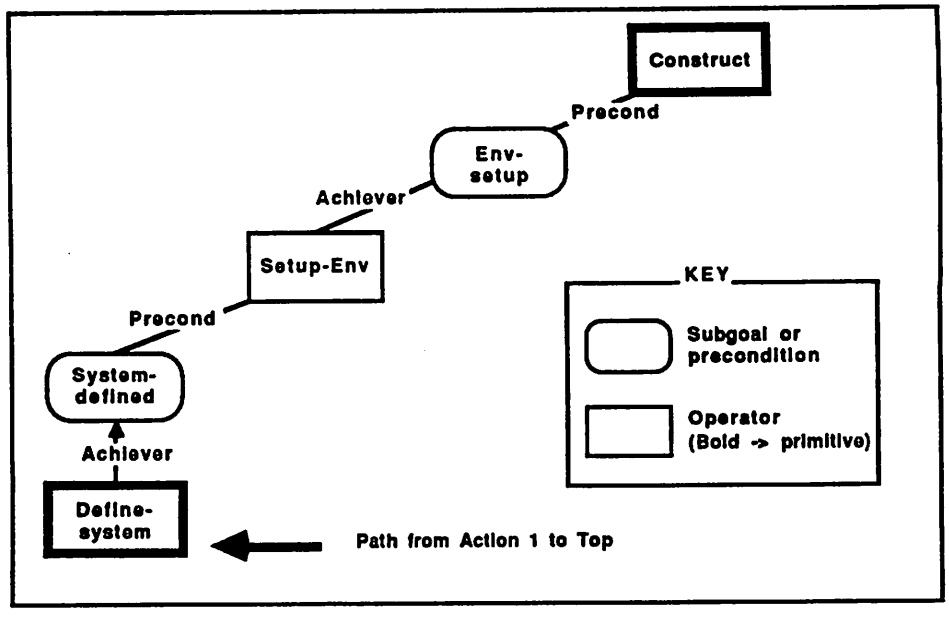

Figure C.3 First Example: Path for Action 1

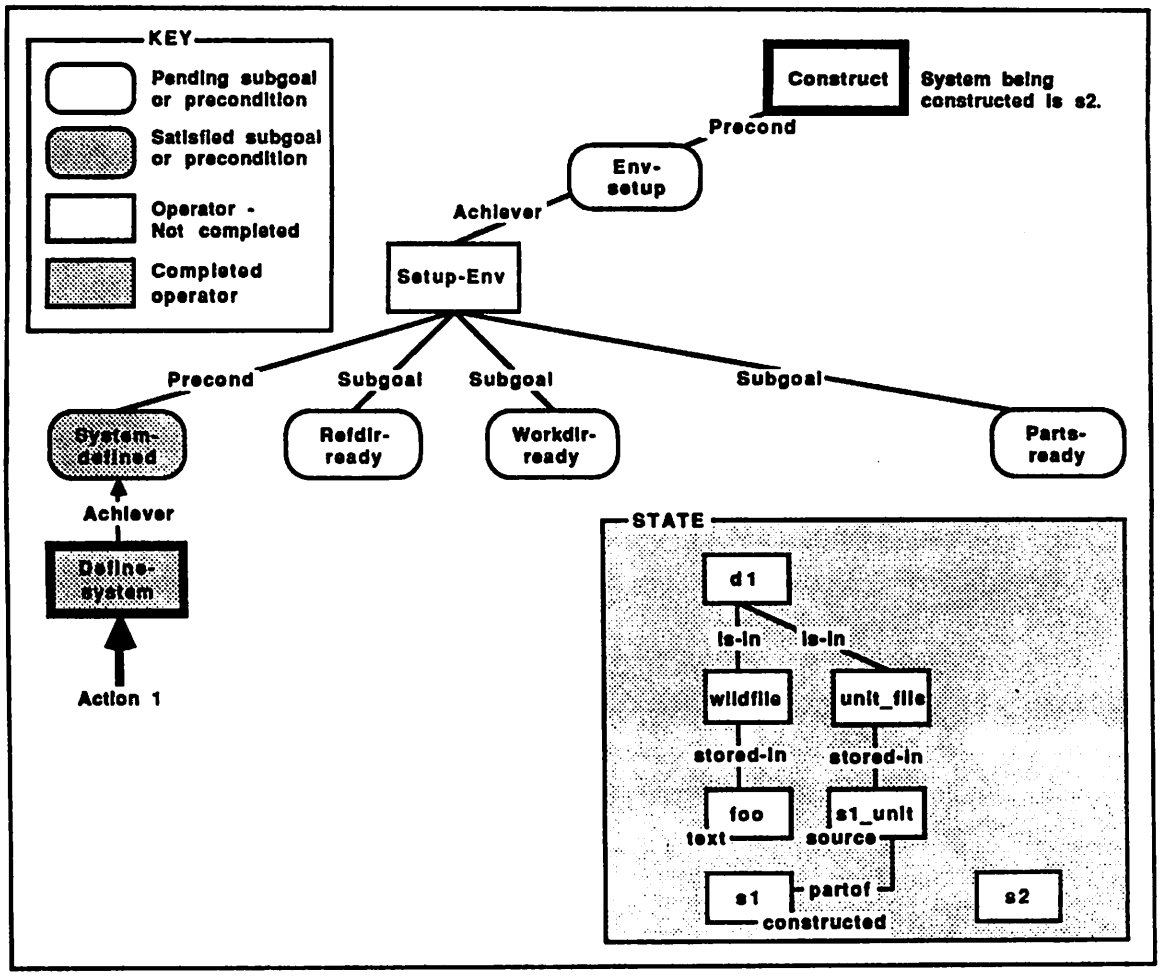

Figure C.4 First Example: Interpretation 2

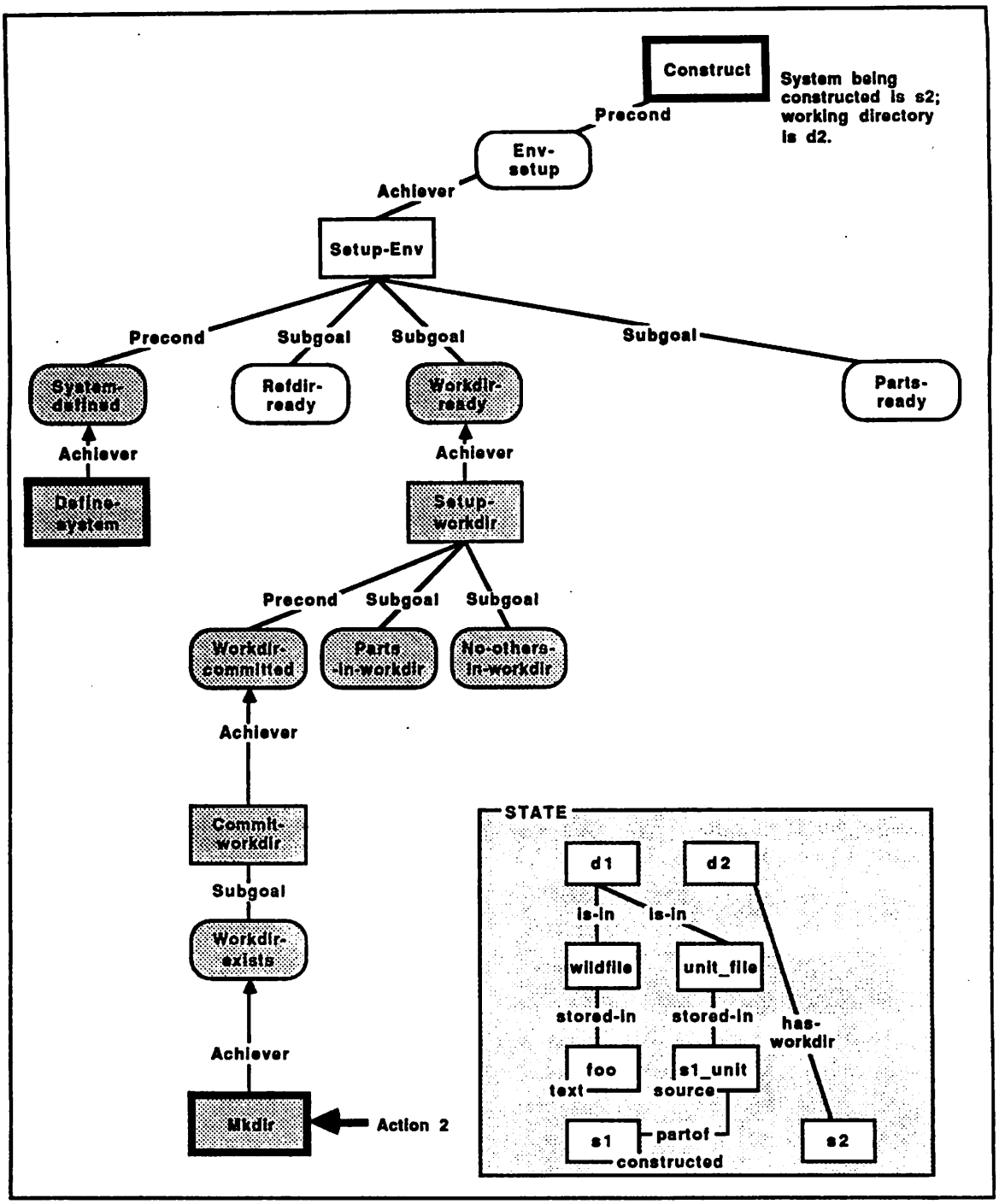

Figure C.5 First Example: Interpretation 3

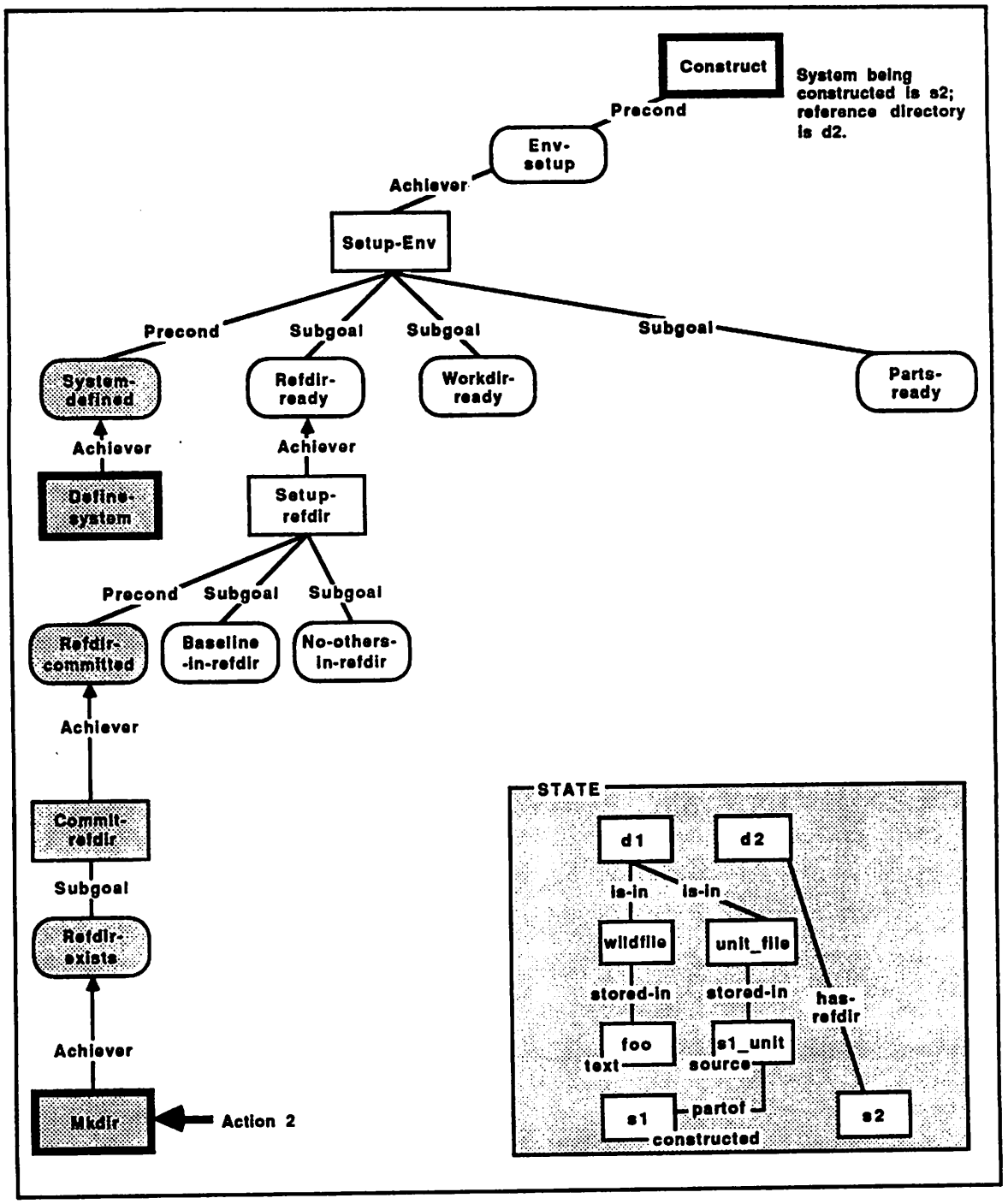

Figure C.6 First Example: Interpretation 4

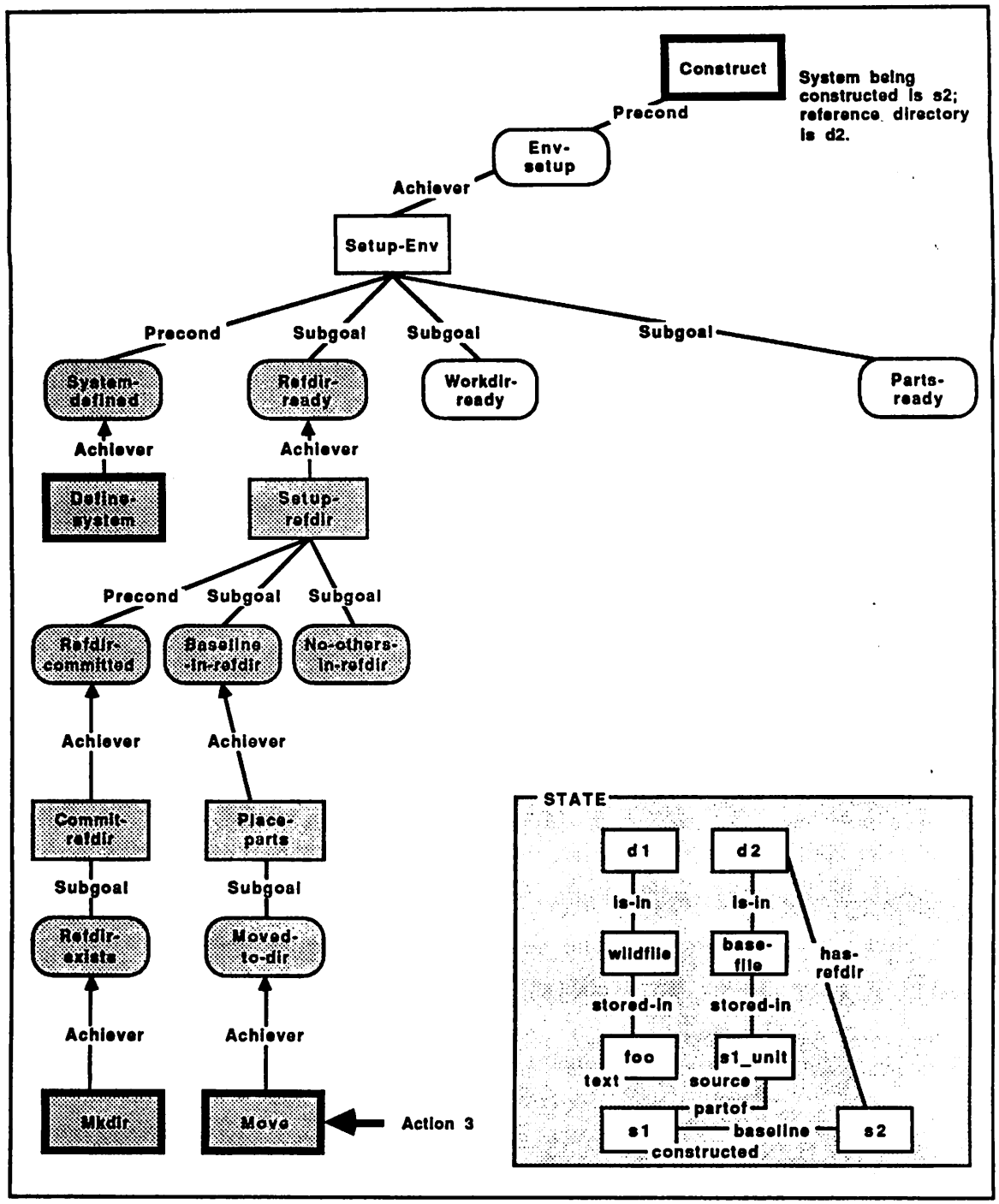

Figure C.7 First Example: Interpretation 5

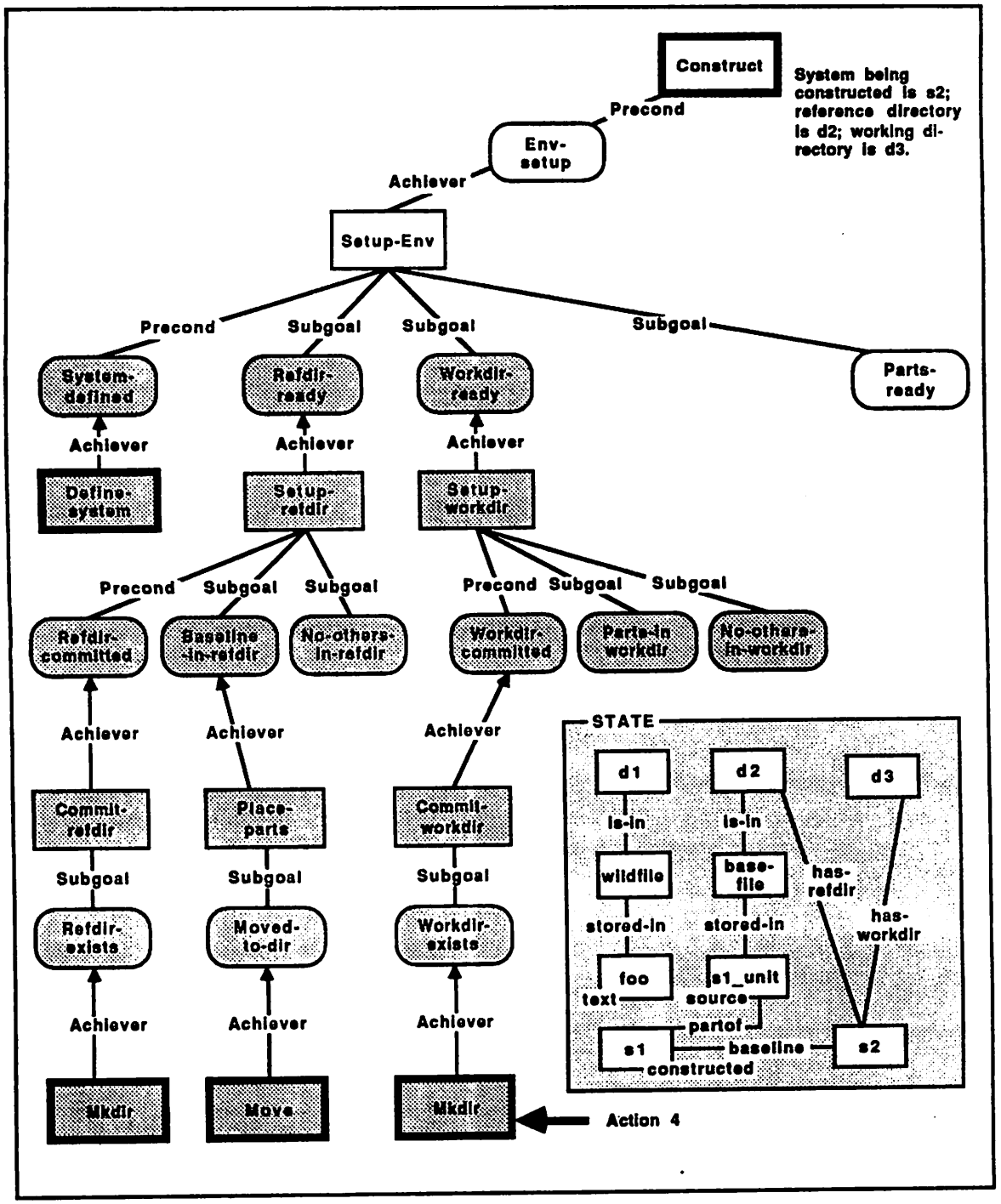

Figure C.8 First Example: Interpretation 7

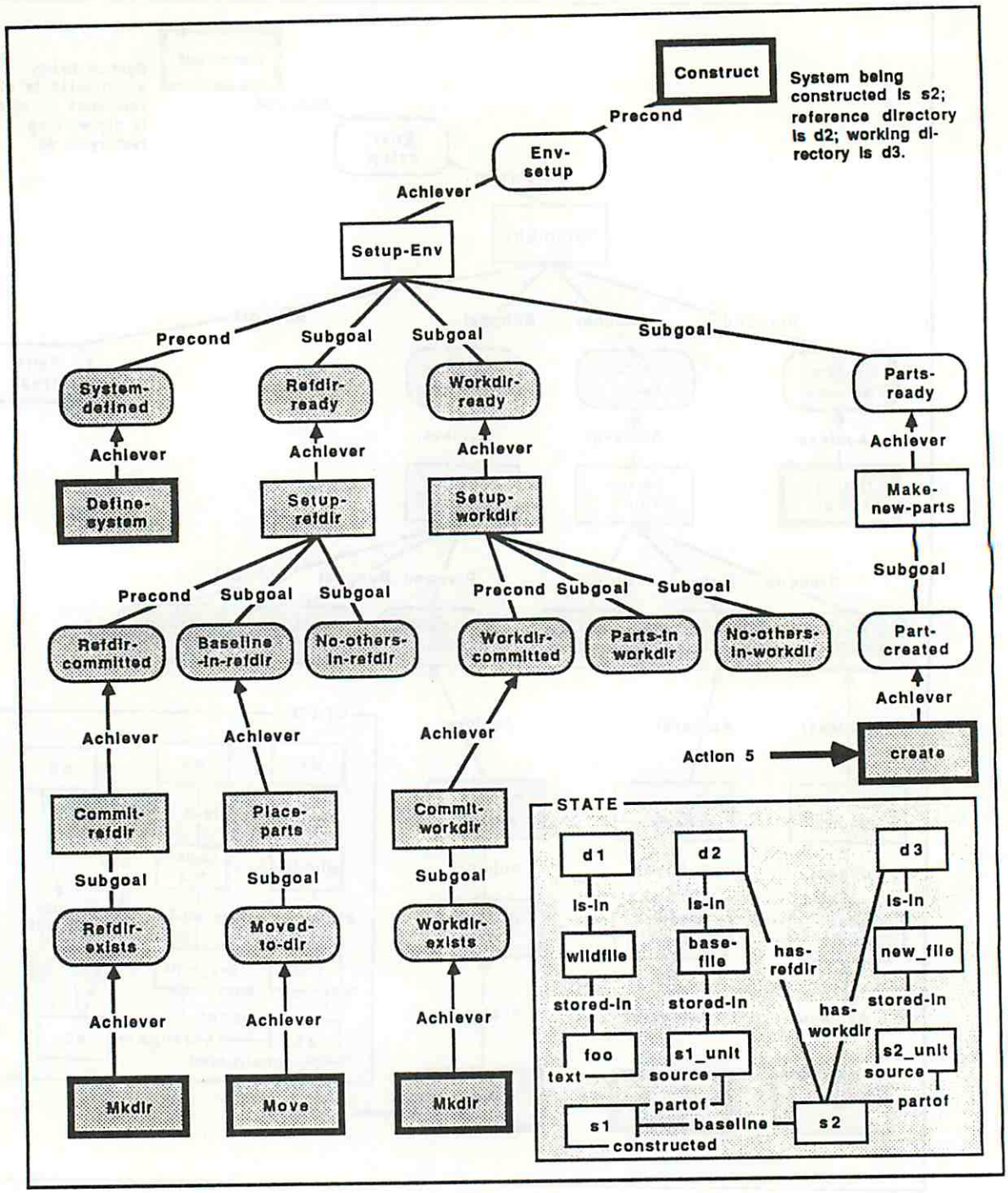

Figure C.9 First Example: Interpretation 8

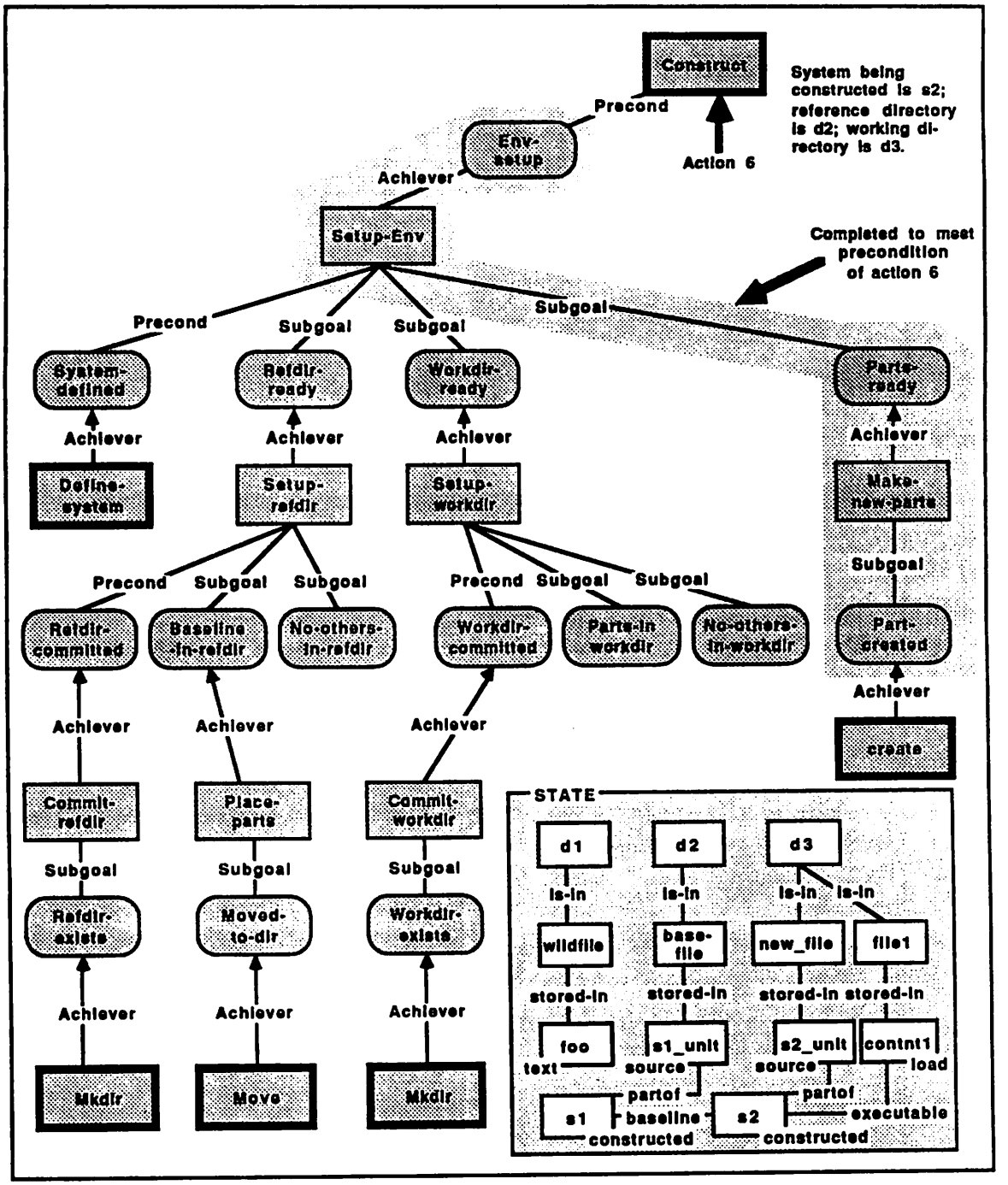

Figure C.10 First Example: Interpretation 9

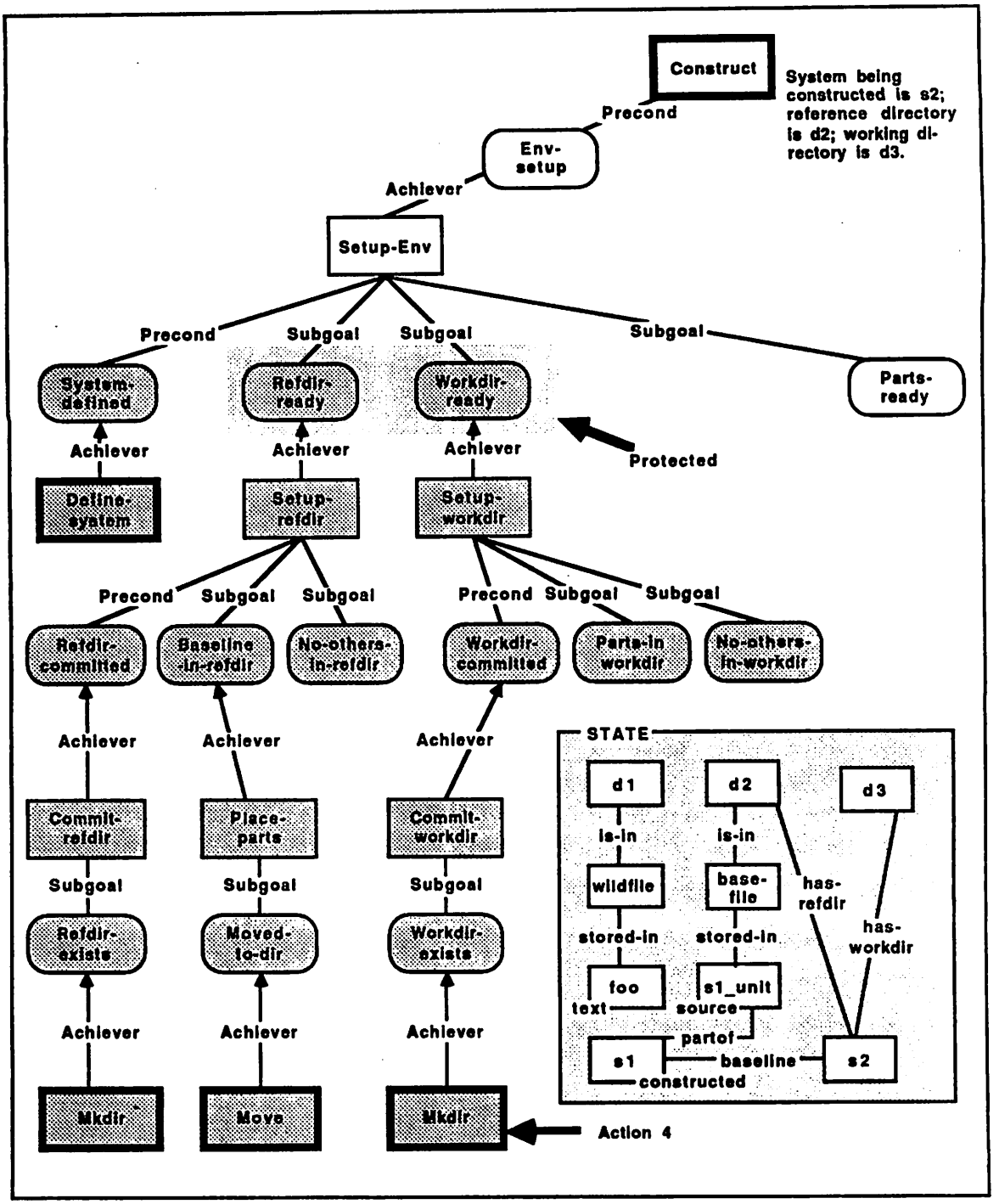

Figure C.11 Second Example: Interpretation 7

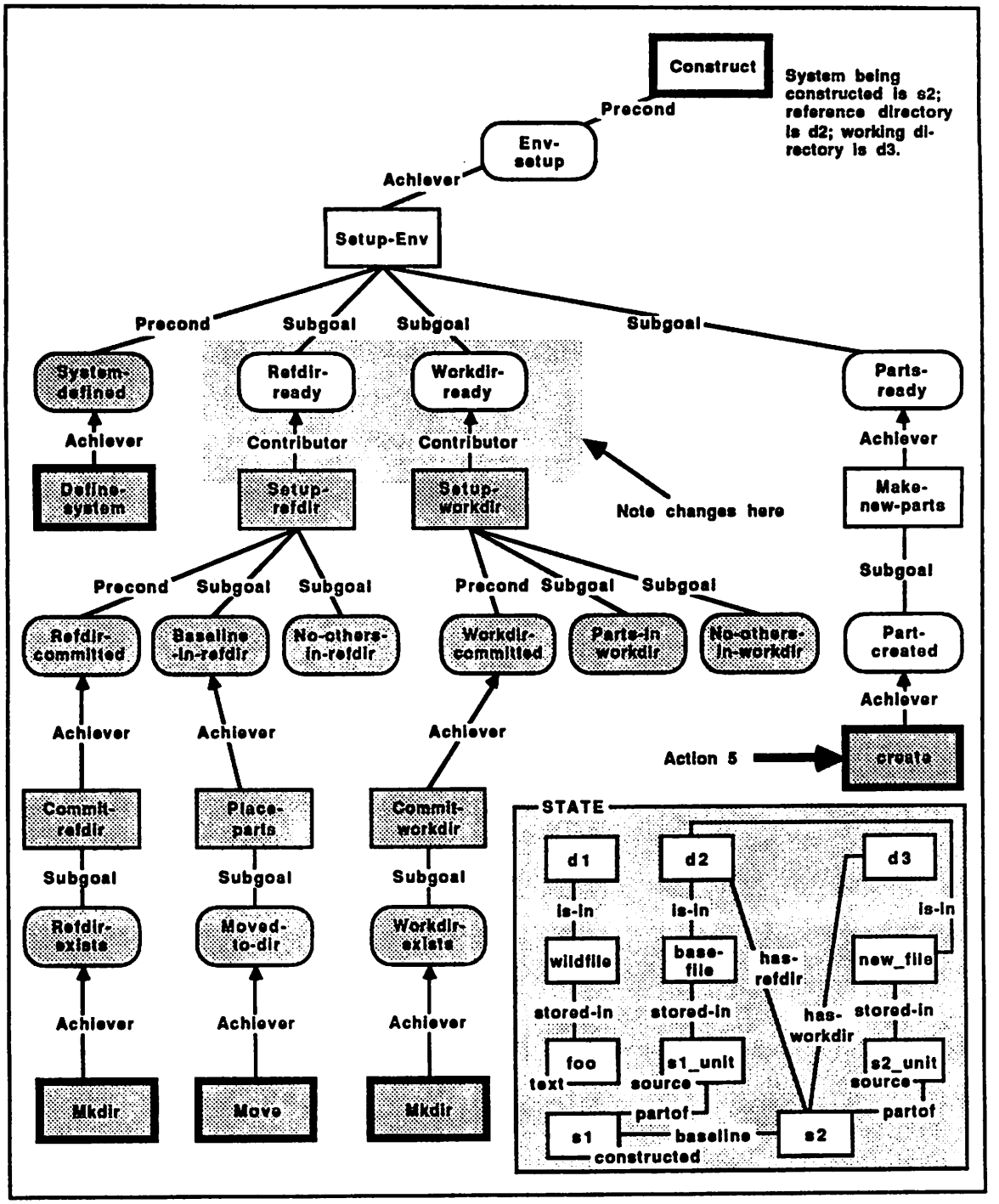

Figure C.12 Second Example: Interpretation 8

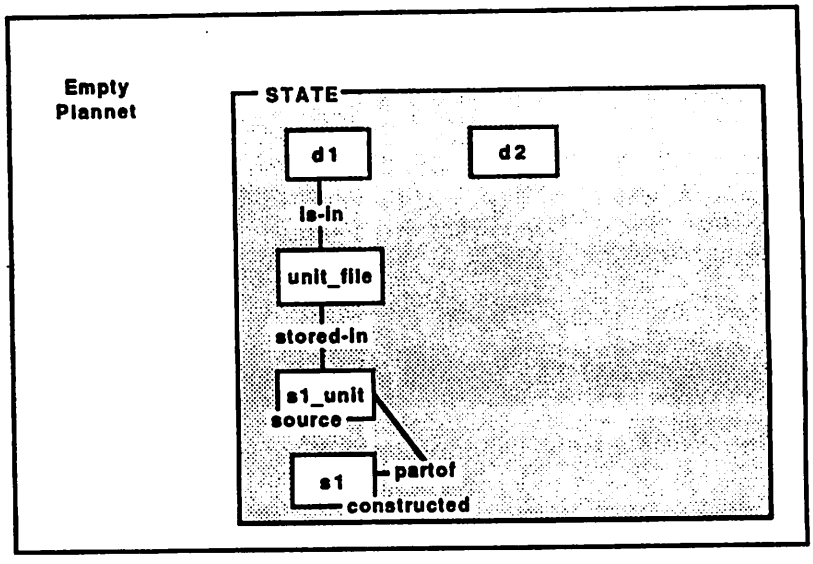

Figure C.13 Third Example: Initial Interpretation

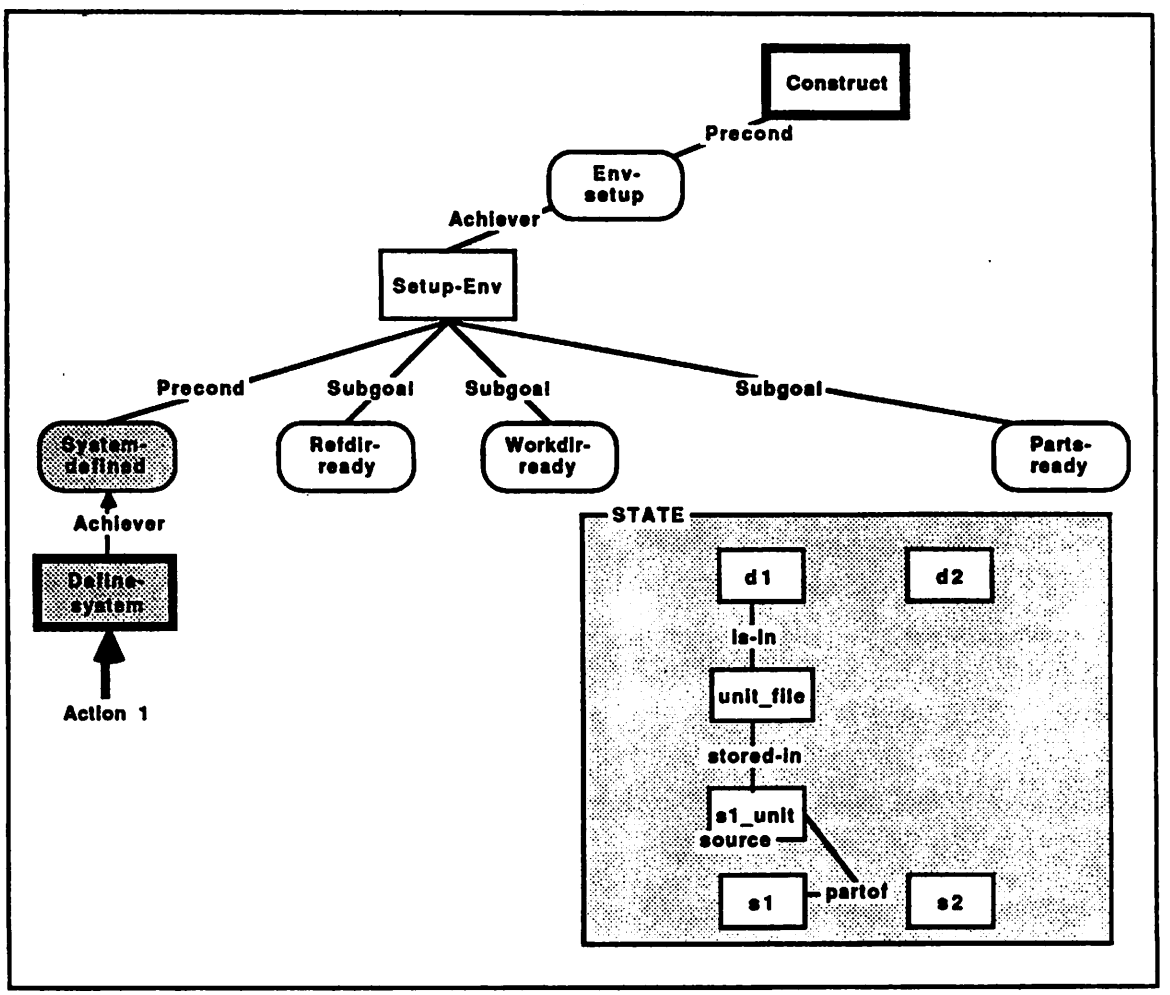

Figure C.14 Third Example: Interpretation 2

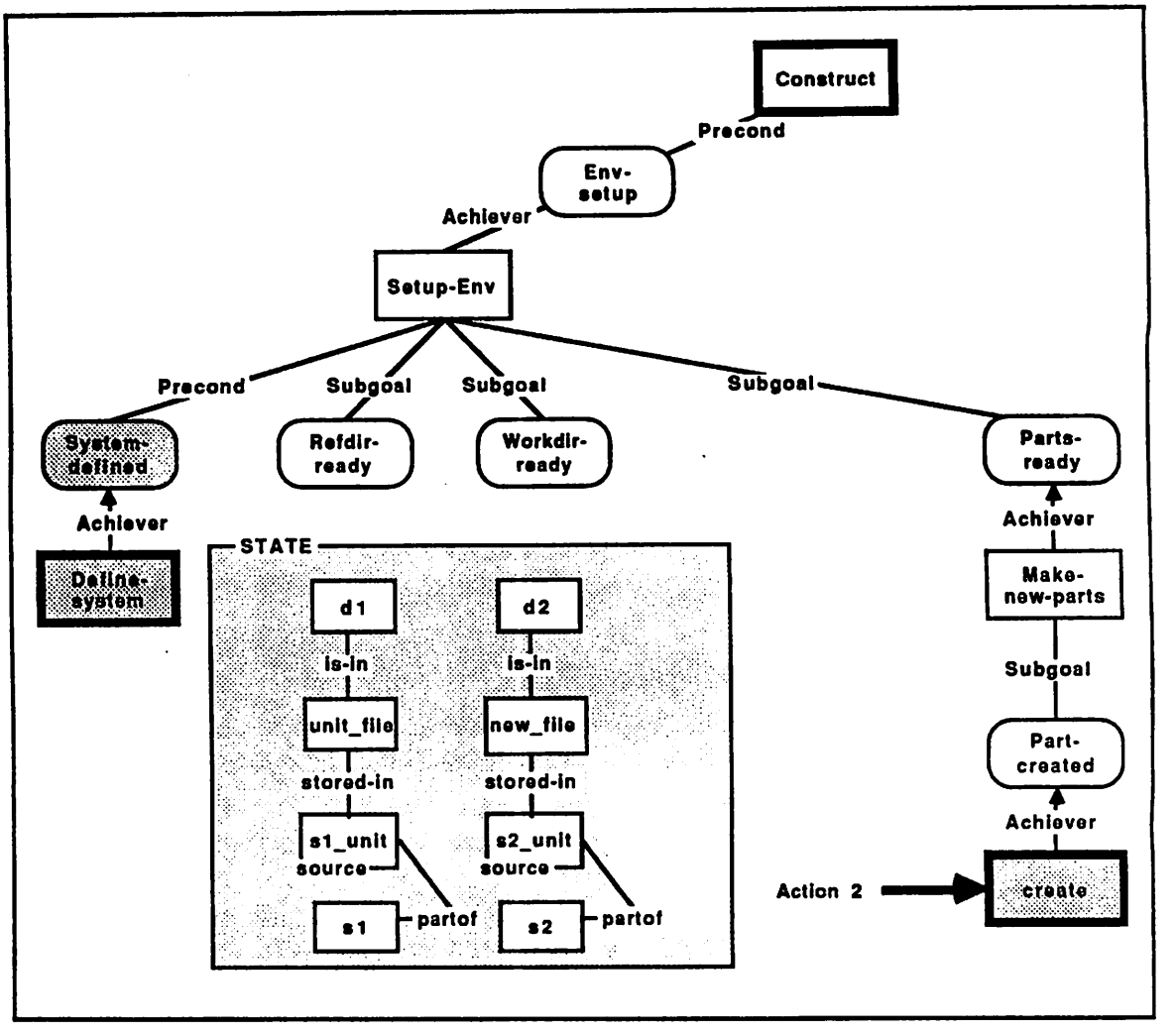

Figure C.15 Third Example: Interpretation 3

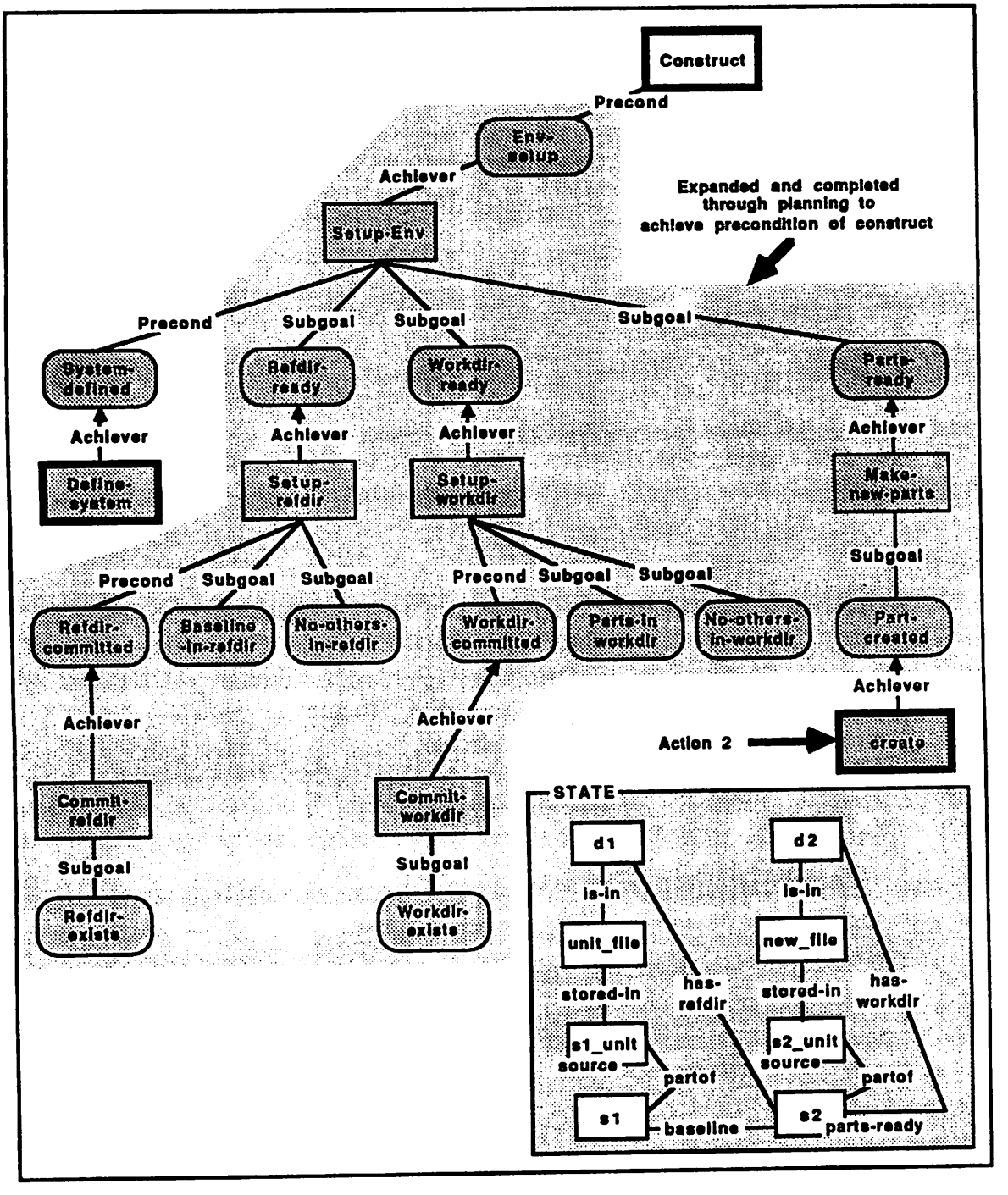

Figure C.16 Third Example: Interpretation 3 Reivsed

APPENDIX D

DESCRIPTION OF TMS IMPLEMENTATION

Data Structures

The state of the TMS is recorded via instances of the relations described below. In a Prolog implementation, these relations are implemented directly as Prolog facts that are dynamically asserted and retracted. In an object-oriented implementation, these relations would be stored as attributes of two different kinds of objects: nodes and justifications. These data structures are internal to the TMS, i.e., they are not directly visible to or modifiable by the problem solver using the TMS.

is_node(node,fact) where
node is a unique name identifying the node
fact is a domain predicate (or its negation) with parameters bound

is_just(just,conclusion,support,exceptions) where
just is a unique name identifying the justification
conclusion is the conclusion node
support is a list (possibly empty) of the support nodes
exceptions is a list (possibly empty) of the exception nodes

A *premise* justification has empty support and exception lists.
A *monotonic* justification has an empty exception list.
A *nonmonotonic* justification has a nonempty exception list.
A justification is in *normal form* if there is only one exception node and it represents the negation of the fact in the conclusion node.
A justification is in *seminormal form* if there are two or more exception nodes, one of which represents the negation of the fact in the conclusion node.

node_conc_in(node, list_of_justs) where
node identifies a node in the TMS
list_of_justs lists all justifications in which *node* is the conclusion

node_hyp_in(node, list_of_justs) where
node identifies a node in the TMS
list_of_justs lists all justifications in which *node* appears in the support list

node_exc_in(node, list_of_justs) where
node identifies a node in the TMS
list_of_justs lists all justifications in which *node* appears in the exception list

node_support(node, list_of_justs) where
node identifies a node in the TMS
list_of_justs is a (possibly empty) list of **all** valid justifications whose conclusion node is *node*.

A *valid* justification is one whose support nodes are all IN and whose exception nodes are all OUT. A node that is the conclusion of at least one valid justification is said to be IN, otherwise it is OUT.

External Interface to TMS

The state of the TMS is maintained and queried through use of the following procedures, which are visible to the problem solving system using the TMS. (These procedures in the external interface make use of internal procedures whose descriptions are given later.)

Add Justification:

Given the conclusion fact, the support facts, and the exception facts:

Instantiate nodes for each of the facts, unless they are already represented by nodes.

Apply the **check for anomalies** (see procedure below).

Generate a new unique justification name, instantiating the justification under this name.

Perform necessary indexing:
Index the conclusion node for the justification in node_conc_in.
Index each support node in the justification in node_support_in.
Index each exception node in the justification in node_exc_in.

Apply **establish support** to the new justification (see procedure below).

Delete Justification:

Given the name of the justification to be deleted:

Apply **remove support** to the justification.

Perform necessary de-indexing:

Remove the conclusion node for the justification from node_conc_in.
Remove each support node in the justification from node_support_in.
Remove each exception node in the justification from node_exc_in.

Delete the relation *is_just* for this justification.

Query

Given a proposition F, the truth value of F is determined and returned. The possible truth values are *certainly true*, *assumed true*, *unknown* (implicitly assumed unknown), *assumed false*, and *certainly false*.

Propositions represented in the TMS can be either IN, OUT or ABSENT. If node(N,F) and node_support(N,List) and List is empty, F is OUT. If node(N,F) and node_support(N,List) and List is not empty, F is IN. If there is no N such that node(N,F), F is ABSENT.

Truth values are computed from IN, OUT and ABSENT by considering the status of both the proposition and its negation. Ignoring for the moment the certainty of propositions, the possible truth values are *true*, *false*, *unknown* and *contradiction*. (The plan recognizer actually uses the TMS in such a way that the truth value contradiction does not occur unless there is an error in the way that the justifications were written. Thus, the plan recognizer deals in a three valued logic involving true, false, and unknown.)

The type of predicate (core or extended) enters into the computation of truth values because, for efficiency reasons, core predicates are stored under a modified closed world assumption. The standard closed world assumption makes it unnecessary to record the truth status of all propositions—propositions not recorded as being true are taken to be false. This means that it is only necessary to instantiate nodes representing non-negated core propositions that are currently true. However, negated core propositions can appear in justifications, and instantiating such a justification requires instantiating a node for the negated core proposition. Therefore, we allow the instantiation of negated core propositions on an "as needed" basis. If such a proposition is true, it must be supported with a premise justification; if it is false, it cannot have a justification, and its negation must be

represented by a node with a premise justification. In the truth table below, entries marked with a * are those affected by the modified closed world assumption. The entries marked with a dash are those cases that do not arise under this assumption.

No special restrictions apply to the representation of nodes for extended state propositions. Thus, the truth table column for extended state propositions is the standard TMS translation from IN/OUT/ABSENT to true/false/unknown/contradiction.

Status (for node(N,F) and node(D, not F))	**Truth of F** (based on type of predicate in F)	
	Core	Extended
N IN, D IN	contradiction	contradiction
N OUT, D IN	false	false
N OUT, D OUT	—	unknown
N IN, D OUT	true	true
N IN, D ABSENT	true	true
N OUT, D ABSENT	—	unknown
N ABSENT, D IN	false	false
N ABSENT, D OUT	—	unknown
N ABSENT, D ABSENT	false*	unknown

Nodes in the TMS are either *certain* or *by-assumption*. Core nodes are certain by definition. An extended state node is certain if it is IN through a premise justification, or through at least one monotonic justification involving only other nodes that are certain. An extended state node that is OUT is certain if its dual is IN and certain. All other extended state nodes are by-assumption. As a result of this definition, a node and its dual (if any) always have the same certainty, and this certainty is reported back with the truth value taken from the table that appears above.

Determine Certainties

Initialize certainties so that no extended state node is certain.

Main loop: For every extended state node that is not certain:

> If there is a valid monotonic justification whose conclusion is this node, and if all the support nodes in this justification are either core nodes or extended state nodes that are certain, then mark this node as certain.

If, during the main loop, new nodes were marked as certain, repeat main loop.
Else, stop.

Notes: This algorithm only marks extended state nodes that are IN. This is the minimal information needed to determine certainty. Core nodes are always certain. An extended state node is certain if it is marked as such (in which case it is IN), or if it is the negation of an extended state node that is marked as certain (in which case it is OUT).

Install Preferences

For each preference rule "Prefer A When B", find an instance where B is true and A is OUT and then do the following:

Save the state of the TMS (by saving the relation *node_support*).

Add a premise justification J whose conclusion is A, thus forcing A IN.

If support for A now includes at least one justification other than J, then delete J (note that A will remain IN.) Also, if B is now false, issue an error message diagnosing a preference that defeats itself.

Else, restore previous state of TMS (by replacing current state of relation *node_support* with saved version).

Repeat until no further changes in the TMS, in case newly installed preferences make it now possible to install preferences that were considered earlier but could not be installed then.

Notes:

It is a fact of life for TMS's that adding a justification followed immediately by deleting that justification does not necessarily return the TMS to its original state. Both the positive and negative sides of this fact are reflected in this algorithm. On the positive side, adding a justification can be used to "reverse" the labels in an even loop; if this succeeds in enabling additional support for a node, deleting the justification leaves the new labels in place. On the negative side, if this attempt at preference installation fails, simply deleting the new justification will not restore the previous TMS state. (The algorithm takes the conceptually simple solution of saving the TMS state beforehand so that it can be restored.)

Example: suppose the TMS has these justifications, A EXCEPT ¬B —> B and EXCEPT B —> ¬B. Let the initial conditions be that both A and ¬B are IN, so B is OUT. Now add a premise justification supporting B (as would be done to install a preference for B); this forces ¬B OUT, allowing B to be supported through the justification involving A. Now delete the premise justification on B. No labels change, because B still has support

through the justification involving A. The net effect is that the loop on B/¬B has been reversed(in this case, all nodes in the loop have labels opposite to their original labels).

The algorithm also depends on the fact that once a preference is installed, it does not have to be continually re-installed. This is due to a property of even loops in the presence of normal and seminormal rules. Once a labelling for the loop is established, that labelling persists until one node in the loop that is IN loses all its valid support, or until a monotonic justification supporting a node in the loop that is now OUT becomes valid. In either case, at that point the preference is no longer applicable.

Passes over the preference rules are made repeatedly until a steady state is reached in order to deal with cases where one preference enables another. Consider the two preferences, Prefer Y when X and Prefer X. If X is initially false, then the preference for Y will not be relevant until the preference for X is successfully installed. These relationships among preferences may not be apparent by just examining the preference rules themselves. Prefer A, Prefer B when C are interrelated when there is a justification A —> C. If the number of passes over the preference rules exceeds the number of preference rules, then there is a pathological interrelationship among the preferences and an error message should be issued. For example, an obvious pathology exists between these preferences which cannot be satisfied if it is possible for both A and C to be true simultaneously: Prefer ¬B when A, Prefer C when ¬B, Prefer B when C. Currently, there is no provision for detecting either benign or pathological interrelationships in advance.

Operations Internal to the TMS

The following procedures, which are not visible to the problem solving system using the TMS, are used to implement the visible operations.

Establish Support:

Given a justification J:

If all the support nodes of J are IN and all the exception nodes of J are OUT, then J is valid and is added to the node support list of its conclusion node CN.

If J is now the only valid justification for CN, that could affect the status of any justification in which CN is a support or exception node, so:

Apply **establish support** to all justifications in which CN is a support node.

Apply **remove support** to all justifications in which CN is an exception node.

Remove Support:

Given a justification J which has just become invalid (or is being deleted).

Remove J from the node support list of its conclusion node CN.

If CN no longer has any valid justifications, that could affect the status of any justification in which CN is a support or exception node, so:

Apply **remove support** to all justifications in which CN is a support node.

Apply **establish support** to all justifications in which CN is an exception node.

Check for Anomalies:

A justification that creates anomalies should not be instantiated. Since odd loops are unsatisfiable and circular arguments are spurious, adding a justification that creates either of these anomalies is not allowed.

Anomalies are defined as follows: let A--->B mean that there is a justification in which A is a support node and B is the conclusion node, and let A-#->B mean that there is a justification in which A is an exception node and B is the conclusion node. The TMS can be thought of as a network of paths from node to node via justifications; these paths are represented by stringing together their individual links:

A--->B and B-#->C yields A--->B-#->C, etc.

An odd loop corresponds to a path from a node back to itself that involves an odd number of -#-> links; a circularity corresponds to a path from a node to itself involving only ---> links. Even loops, which are allowed, are satisfiable but lead to multiple labellings of the TMS; in fact, rules in normal and seminormal form always create even loops.

Anomalies are discovered by considering paths in reverse order, starting from the conclusion node of the proposed new justification, working "backward" to its support and exception nodes, and thence to justifications involving those nodes as conclusions, etc. During this backwards tracing, a count of occurrences of the two different types of links is maintained; if a

path encounters the original conclusion node again, the counts are checked to determine if an anomaly has been detected.

APPENDIX E

REVISED ALGORITHM FOR PLAN RECOGNITION

NOTE: Additions and changes to the algorithm are denoted with underlining.

Given: one interpretation (plan network and corresponding world state) in the interpretation tree that is the current "focus" of attention. Assume this interpretation covers actions 1 through N-1.

START: Get action N.

Generate candidate interpretations for N:

Finding all possible paths to account for action N uses the pre-computed reachability information. This information is stored in the form:

reachable (op1,part1,op2,op3, part3),

where op1, op2 and op3 are operators, part1 is a precondition or subgoal of op1, and part3 is a precondition or subgoal of op3. This means that there is a path from op1/part1 to primitive operator op2 and a possible first step on the path (downward from op1 to op2) is the step to op3/part3. If there are no intervening steps between op1/part1 and op2, then op3/part3 are omitted.

Initialize the set PATHS of paths as follows. (Each step on a path will be represented by an operator paired with a specific precondition or subgoal of that operator. Paths start at top level operators and step "downwards", ending eventually at a primitive operator.)

Consider all operator instances O in the current interpretation and all their instantiated (e.g., pending) preconditions P or subgoals S. For each combination of O/P and/or O/S that is reachable from action N, the existing path from the top level operator TOP to O/P or O/S is an element of S. These elements of PATHS are

extension paths. The dynamic namescopes of these paths are the dynamic namescopes of their respective TOPs.

Now consider all top level operators, regardless of whether instantiated. For each top level operator O and each of its preconditions P or subgoals S that are reachable from action N, make a (one-step) path consisting of O/P or O/S in PATHS. These elements of PATHS start new plans. Initialize a new dynamic namescope for each of these paths, assigning new dynamic names to the parameters of O.

Now extend the paths in PATHS all the way to action N. Remove from PATHS each path P that does not end with a primitive operator (equal to action N), and do the following:

Let O1/PS1 be the last step on path P. For each O3/PS3 such that:
reachable(O1,PS1, Action N, O3, PS3),
add a path to PATHS that consists of P extended to include O3/PS3 as the last step.

Retrieve the static-to-static namemap associated with using O3 to achieve PS1 of O1 (if there are two or more possible namemaps, make as many copies of the path as there are namemaps and repeat the name calculations using a different namemap on each path). Use this namemap, and the static-to-dynamic namemap of O1 to compute the static-to-dynamic namemap of O3, assigning new dynamic names to any parameters of O3 that are not otherwise mapped.

Prune candidates:

Check each path P in PATHS as follows:

If P extends an existing plan, record the bindings of action N under the appropriate dynamic names for the plan and reject P if binding inconsistencies are detected.

If P starts a new plan, then initialize the appropriate dynamic names for that plan with the bindings of action N.

Traverse P from action N to the top, considering each operator O on the path:

Check preconditions of O:
If path P reaches O through a precondition of O, or if O has already started, do nothing. Otherwise, check that all the preconditions of O are satisfied. If any precondition check returns certainly false, make a note of that fact. Initialize the number of violated assumptions to the number of preconditions returning unknown or assumed false (for later use in computing credibility).

Check constraints of O:
Omit constraint checks entirely if O has already started. Omit consideration of a constraint if path P reaches O through a

precondition of O and some parameters in the constraint are not bound. Test all other constraints, rejecting P if any tested constraint is certainly false. Add the number of constraints returning unknown or assumed false to the number of violated assumptions.

Check goal of O:

If there are bindings for all parameters in the goal of O, then check to see that the goal is not already true. (Note that goals of operators do not involve extended state predicates and hence evaluate either to certainly true or certainly false.) If this check fails, reject P.

Check to see that the goal of the top level operator in P does not duplicate the goal of the top level operator in any other existing plan. Omit this check if there are any unbound parameters in the goal at the top of P. Reject P if the check is made and fails.

Instantiate a new interpretation that incorporates P (unless P has already been rejected). Let I be the interpretation (at level N-1) that is the current focus; I will be the parent of the new interpretation. Perform the instantiation by duplicating I (both plan network and world state), then adding to I the new part of the path P and new bindings (taken from action N or discovered as part of processing P).

Compute the credibility of the new interpretation as zero minus the number of violated assumptions.

Focus among interpretations at level N:

If there are no interpretations at level N (or if all such interpretations have been rejected), inform user of possible error in action N. User may choose to perform action N anyway (causing recognizer to work harder to find an interpretation for action N) or to perform a different action. Depending on user response, either:

Reject parent interpretation at level N-1, and then backtrack (chronologically) as follows:

Re-focus by tracing back through interpretation tree to nearest ancestor with at least one remaining child interpretation (call the level of the child level J). Select from among competing children if necessary, and update status of selected interpretation SI (see algorithm below).

Start cycle over on action J+1 with new focus SI, continuing cycles until all actions including N are accounted for.

Or, terminate cycle (delete all interpretations at level N, then return to START and wait for new action N).

Otherwise, select among interpretations on the basis of maximum credibility. Use the other heuristics to break ties if necessary. Make a note of the credibility level of the second choice. Replace current focus (at level N-1) with selected focus (at level N). (The heuristics imply that, if possible, the selected interpretation will involve a path P that extends an existing plan and will not require planning to satisfy any preconditions.)

Update status of new focus (at level N):

If the path P on which this interpretation was constructed involved any failed preconditions (i.e., those evaluating to certainly false), call planner to achieve them all. If planning fails, reject this interpretation; repeat focusing at level N, as described above. The planner may discover additional violated assumptions and call reconciliation to handle them. This affects the credibility of this interpretation, and it may fall below the credibility of the second choice. If this happens, re-focus on the second choice but do not reject the interpretation whose credibility slipped during planning. (Re-focusing causes all alternatives to be re-ranked, and there may be a new second choice.) Continue by updating the status of the interpretation now in focus.)

Call reconciliation to revise assumptions, if needed. If reconciliation fails, reject this interpretation and repeat focusing at level N, as described above.

Remove from protection list the preconditions of all operators that "started" with this action (e.g., all operators whose preconditions were actually tested during processing of path P).

Traverse the path P from action N to the top:

At each operator O:
Check for completion (e.g., goal of O certainly true).
If completed, post effects of operator via TMS premises.

At each precondition or subgoal:
Check for satisfaction (e.g., precondition or subgoal certainly true).
If satisfied, put on protection list.

Halt traversal at first incomplete operator, or unsatisfied precondition or subgoal.

Complete computation of new world state by:
Generating new instances of justifications.
Computing node certainties.
Installing preferences among justifications.

Process pending conditions:

Check each pending (dynamic) precondition:
If now certainly true, put precondition on protection list.

If no dynamic preconditions of operator are now certainly false, instantiate the subgoals of operator.

Check each pending subgoal:

If now certainly true, put subgoal on protection list.
If all subgoals of operator are now certainly true, post effects of operator via TMS justifications and take subgoals off protection list.

Repeat processing of pending conditions until no further effects are posted.

Check protection list for violations:

(Note that an expression had to be certainly true when it was originally put on the protection list.)

Find all conditions (preconditions or subgoals) on protection list that are no longer certainly true. If there are no violations, then updating is complete.

Otherwise, inform user of violations. User has three options: to perform action in spite of the violations, to take a different action, or to force the recognizer to find another interpretation of the action (in which presumably there are no protection violations.)

Depending on user response, either:

Take conditions off protection list, and mark them unsatisfied.

Or, terminate cycle, deleting all interpretations at level N, changing focus back to parent interpretation at level N-1, and returning to START to wait for new action N.

Or, reject this interpretation. If no remaining interpretations at level N, backtrack; otherwise, focus at level N again.

APPENDIX F

ALGORITHM FOR RECONCILIATION

Given: a list L of nodes to be brought IN. Let ACTION be the number of the action currently being recognized. The algorithm is given first, followed by supporting procedures.

Handle trivial case:
If L is empty or if every node on L is already IN, return with success.

Analyze-changes:
Call **all-in-and-out** with L as the list of nodes to be brought IN and an empty list of nodes to be brought OUT, and 0 as the cycle number. This returns a list of alternatives.

Rate-alternatives:
For each alternative returned by all-in-and-out, remove duplicates (a node listed two or more times on the IN list, justifications listed two or more times on either the BLOCK or UNDO lists.) It is not necessary to process the REVERSE list for duplicates.

Rate the alternatives by counting +1 for each node on the IN list, +1 for each justification on the BLOCK list, and -1 for each justification on the UNDO list. Ignore entries on the REVERSE list.

Let MaxP be the maximum rating among the alternatives.

Save the current state of the TMS.

Select-alternative:
Select alternative with lowest rating. If there are no alternatives, quit, reporting failure.

Install-alternative:
For each node on the IN list, create a dummy justification. (The predicate for the new node in the dummy justification captures the fact that this justification was created for action ACTION.)

For each justification on the BLOCK list, delete the justification and re-create it with an extra node (the blocking node) on the support list. (The predicate for the new node in the blocked justification captures the fact that this justification was blocked for action ACTION.)

For each justification on the UNDO list:
If it is a dummy justification, delete it.
If it is a blocked justification, delete and recreate it without the blocking node.

Install preferences.

Check for success:
Count the number N of nodes on the list L that are not now IN.
Count the number C of contradictions (node and its negation both IN).

Decide if more to do:
If N+C = 0, return with success.
Else if the rating of the alternative is less than or equal to MaxP:
If N>0, discard this alternative. If that leaves no alternatives, create an alternative with every node from L on the IN list and all other lists empty. Rate this new alternative as MaxP + 1.
If N=0, re-rate this alternative as MaxP + C
Else if the rating of the alternative is greater than MaxP:
Call **all-in-and-out** with the IN list empty and all contradictions on the OUT list and ACTION as the cycle number. **Rate** the alternatives, **install** each alternative in turn, discard this candidate if none of the alternatives meets the **check-for-success** or if there are no alternatives.

Restore the saved state of the TMS.
Repeat from **select-alternatives.**

All-in-and-out:

Given a list LIN of nodes to be brought IN and a list LOUT of nodes to be brought OUT and a cycle number CYC, initialize answer to an empty list of alternatives.

For each node on LIN that is not already IN, call **node-in**, giving a null path. **And** the result onto the current answer. For each node on LOUT that is not already OUT, call **node-out**, giving a null path. **And** the result onto the current answer.

Node-in:

Let N be the node to be brought IN and P be the path by which N was reached. (N is not already IN.) The default answer consists of a single alternative with N on the IN list and the other lists empty

Initialize the answer to the empty list.
For each justification J whose conclusion is N:

Ignore J if it is in otherwise form. Otherwise, call **validate-just** on J giving P union {N} as the path, **or**'ing the result onto the answer. If validate-just fails on some J, ignore that justification.

If the current answer is empty and there is no J in otherwise form, set the current answer to the default answer.

Now, deal with ¬N as follows. If ¬N is OUT, return the current answer. Otherwise, there are six alternative cases:

(1) If ¬N is IN and on the path P and the current answer is the empty list, return failure.

(2) If ¬N is IN and on the path P and the current answer is the default answer and there is an unconditional preference for N, then return the default answer.

(3) If ¬N is IN and on the path P and the current answer is the default answer and there is no unconditional preference for N, then return as the answer a single alternative with N on the IN list and ¬N on the REVERSE list.

(4) If ¬N is IN and on the path P and the current answer is nonempty and not the default answer, then return it.

(5) If ¬N is IN via nonmonotonic rules only and not on the path P and there is an unconditional preference for N, return the current answer.

(6) Else, call **node-out** on ¬N, giving P union {N} as the path, **and**'ing the result with the answer, and returning that value. If there is an unconditional preference for N, then node-out should be restricted to monotonic justifications only. If node-out fails, return failure.

Node-out:

Let N be the node to be brought OUT and P be the path by which N was reached. (N is not already OUT.) Let F be a flag set to true if consideration is restricted to monotonic justifications only.

> Initialize the answer to the empty list. Find the set K of all currently valid justifications whose conclusion is N; if consideration is limited to monotonic justifications only, remove all nonmonotonic elements of K. For each remaining element J of the set K:
>
> > Call **invalidate-just** on J giving P union {N} as the path, **and**'ing the result the current value of the answer. Return failure if **invalidate-just** fails.
>
> Return the current answer.

Invalidate-just:

Let J be the justification that is to be invalidated, if possible; let N be the conclusion of J. Let P be the path by which J was reached, with N at the end of the path.

> If J is a dummy justification created for an action other than CYC, then return as the answer one alternative in which J is on the UNDO list and all other lists are empty. If J is a dummy justification created for action CYC, then return failure.
>
> Otherwise, find the support nodes of J that are IN by assumption, calling them the set S, and find the exception nodes of J that are OUT by assumption, calling them the set E. Exclude ¬N from E.
>
> Remove from S and E any node that appears on the path P.
>
> Initialize the answer to the empty list. Then:
>
> > Call **node-out** on each member of S giving P as the path, **or**'ing the result with the current answer. If node-out fails on a member of S, simply ignore that member.
> >
> > Call **node-in** on each member of E giving P as the path, again **or**'ing the result with the current answer. If node-in fails on a member of E, simply ignore that member.
>
> If J is nonmonotonic, deal with ¬N as follows. If ¬N is on the path P, do nothing. If ¬N is not on the path P, call **node-in** on ¬N giving P as the path. If node-in fails, do nothing. If node-in succeeds, and if the result includes any

alternative that has N on the REVERSE list, then modify each such alternative as follows: remove N from the REVERSE list and add J to the BLOCK list. (Note: the modification should be omitted if there is an unconditional preference for ¬N.) Then, **or** the modified result of node-in (changed and unchanged alternatives inclusive) onto the current answer.

If the answer is no longer the empty list, return the current value of the answer. If the current answer is empty and J is nonmonotonic, return as the answer a single alternative consisting of J on the BLOCK list and all other lists empty. Otherwise, return failure.

Validate-just:

Let J be the justification that is to be made valid, if possible; let N be the node that is the conclusion of J. Let P be the path by which J was reached, where N is the last entry on the path.

If J has the keyword *not-validatable*, return failure.

If J is a blocked justification that was blocked for action CYC, return failure.

If any support node of J is OUT with certainty or any exception node of J is IN with certainty, return failure (J is not activatable).

Find the support nodes of J that are OUT by assumption, calling them the set S, and find the exception nodes of J that are IN by assumption, calling them the set E. If there is a blocking node among the support nodes, do not include it in the set S. Also, exclude ¬N from E. (This means that validate-just returns an incomplete answer on nonmonotonic justifications, which is made complete by special code in node-in.)

If any node in either S or E appears on the path P, return failure.

If S and E are empty, return as the answer a single alternative consisting of ¬N on the REVERSE list and the other lists empty (unless J is currently blocked, in which case J is on the UNDO list).

Initialize the answer to the empty list. Then:

Call **node-in** on each member of S giving P as the path, **and**'ing the result onto the current answer. If node-in fails, then return failure.

Call **node-out** on each member of E giving P as the path, again **and**'ing the result onto the current answer. If node-out fails, then return failure.

If J is nonmonotonic and ¬N is on the path P, then add ¬N to the REVERSE list of each alternative in the current answer. Omit this modification when there is an unconditional preference for N.

If J is not blocked, return the current answer. If J is blocked, add J to the UNDO list of each alternative in the current answer, returning the result.

AND

Given two lists of alternatives, where an alternative consists of a list of 4 lists (INs, BLOCKs, UNDOs, and REVERSEs):

If one of the two lists of alternatives is empty or consists of a single alternative with 4 empty lists, return the other list as the answer.

Otherwise, form a new list with one list element as for each possible combination of one alternative A1 from the first list with one alternative A2 from the second list. The combination consists of a list of four lists, each formed by appending one of the four lists in A2 to the corresponding list in A1.

OR

Given two lists of alternatives, where each alternative consists of a list of 4 lists (INs, BLOCKs, UNDOs, and REVERSEs):

Append the two lists (thereby forming one list containing all alternatives), returning the resulting list as the answer.

BIBLIOGRAPHY

[Allen, 1983]
Allen, J.F. "Recognizing Intentions from Natural Language Utterances." *Computational Models of Discourse*. Edited by M. Brady. Cambridge, MA: MIT Press, 1983.

[Allen, 1984]
Allen, J.F. "Towards a General Theory of Action and Time." *Artificial Intelligence*, 23(1984), 123-154.

[Azarewicz et al., 1986]
Azarewicz, J. et al. "Plan Recognition for Airborne Tactical Decision Making." *Proceedings AAAI-86*, Palo Alto, CA: Morgan Kaufmann, 1986, 805-811.

[Balzer, 1987]
Balzer, R.M. "Living in the Next Generation of Operating System." *IEEE Software*, November 1987, 77-85.

[Balzer et al., 1983]
Balzer, R.M.; Cheatham, T.E.; and Green, C. "Software Technology in the 1990's: Using a New Paradigm." *IEEE Computer*, November 1983, 39-45.

[Bernstein, 1987]
Bernstein, P.A. "Database System Support for Software Engineering." *Proceedings Ninth International Conference on Software Engineering*, Washington, D.C.: IEEE Computer Society Press, March 1987, 166-178.

[Broverman & Croft, 1985]
Broverman, C.A. and Croft, W.B. "A Knowledge-based Approach to Data Management for Intelligent User Interfaces." *Proceedings of Conference for Very Large Databases 11*, New York: ACM Press, 1985, 96-104.

[Broverman & Croft, 1987]
Broverman, C.A. and Croft, W.B. "Reasoning About Exceptions During Plan Execution Monitoring." *Proceedings AAAI-87*, Palo Alto, CA: Morgan Kaufmann, 1987, 190 -195.

[Broverman et al., 1986]
Broverman, C.A.; Huff, K.E.; and Lesser, V.R. "The Role of Plan Recognition in Intelligent Interface Design." *Proceedings of Conference on Systems, Man and Cybernetics*, Washington, D.C.: IEEE Computer Society Press, 1986, 863-868.

[Bisiani et al., 1988]
Bisiani, R.; Lecouat, F.; and Ambriola, V. "A Planner for the Automation of Programming Environment Tasks." *Proceedings Twenty-first International Hawaii Conference on System Sciences*, Washington, D.C.: IEEE Computer Society Press, 1988.

[Carver, 1988]
Carver, N. "Evidence-based Plan Recognition." Technical Report 88-13, Department of Computer and Information Sciences, University of Massachusetts, Amherst, MA, 1988.

[Carver et al., 1984]
Carver, N.; Lesser, V.R.; and McCue, D. "Focusing in Plan Recognition." *Proceedings AAAI-84*, Palo Alto, CA: Morgan Kaufmann, 1984, 42-48.

[Chapman, 1985]
Chapman, D. "Nonlinear Planning: A Rigorous Reconstruction." *Proceedings IJCAI-85*, Palo Alto, CA: Morgan Kaufmann, 1985, 1022-1024.

[Chapman, 1987]
Chapman, D. "Planning for Conjunctive Goals." *Artificial Intelligence*, 32:3 (July 1987), 333-377.

[Charniak, 1986]
Charniak, E. "A Neat Theory of Marker Passing." *Proceedings AAAI-86*, Palo Alto, CA: Morgan Kaufmann, 1986, 584-588.

[Charniak et al., 1980]
Charniak, E.; Reisbeck, C.K.; and McDermott, D.V. *Artificial Intelligence Programming*. Hillsdale, NJ: L. Erlbaum Associates, 1980.

[Chen, 1976]
Chen, P.P. "The Entity-relationship Model: Toward A Unified View of Data." *ACM Transactions on Database Systems*, 1:1 (March 1976), 9-36.

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

[Croft & Lefkowitz, 1988]
Croft, W.B. and Lefkowitz, L.S. "Knowledge-based Support of Cooperative Work." *Proceedings Twenty-first International Hawaii Conference on System Sciences*, Washington, D.C.: IEEE Computer Society Press, 1988, 312-318.

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

[Czuchry & Harris, 1988]
Czuchry, A.J. and Harris, D.R. "KBRA: A New Paradigm for Requirements Engineering." *IEEE Expert*, 3:4 (Winter 1988), 21-35.

[Dean & Boddy, 1988]
Dean, T. and Boddy, M. "An Analysis of Time-Dependent Planning." *Proceedings AAAI-88*, Palo Alto, CA: Morgan Kaufmann, 1988, 49-54.

[DeJong & Mooney, 1986]
DeJong, G.F. and Mooney, R.J. "Explanation-Based Learning: An Alternative View." *Machine Learning*, 1:2 (April 1986), 145-176.

[deKleer, 1986]
deKleer, J. "An Assumption-based TMS." *Artificial Intelligence*, 28 (1986), 127-162.

[deKleer, 1988]
deKleer, J. "A General Labelling Algorithm for Assumption-Based Truth Maintenance." *Proceedings AAAI-88*, Palo Alto, CA: Morgan Kaufmann, 1988, 188-192.

[deKleer & Williams, 1986]
deKleer, J. and Williams, B.C. "Back to Backtracking: Controlling the ATMS." *Proceedings AAAI-86*, Palo Alto, CA: Morgan Kaufmann, 1986, 910-917.

[Dowson, 1987]
Dowson, M., ed. *Proceedings of the Third International Software Process Workshop*. Washington, D.C.: IEEE Computer Society Press, 1987.

[Doyle, 1979]
Doyle, J. "A Truth Maintenance System." *Artificial Intelligence*, 12 (1979), 231-272.

[Etherington, 1987]
Etherington, D.W. "Formalizing Nonmonotonic Reasoning Systems." *Artificial Intelligence*, 31 (1987), 41-85.

[Feldman, 1979]
Feldman, S.I. "MAKE—A Program for Maintaining Computer Programs." *Software Practice and Experience*, 9:4 (April 19790, 255-265.

[Fickas, 1985]
Fickas, S.F. "Automating the Transformational Development of Software." *IEEE Transactions on Software Engineering*, 11:11 (November 1985), 1268-1277.

[Fikes & Nilsson, 1971]
Fikes, R.E. and Nilsson, N.J. "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving." *Artificial Intelligence*, 3 (1971), 189-208.

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

[Genesereth & Nilsson, 1987]
Genesereth, M.R. and Nilsson, N.J. *Logical Foundations of Artificial Intelligence*. Palo Alto, CA: Morgan Kaufmann, 1987.

[Goodwin, 1982]
Goodwin, J. "An Improved Algorithm for Nonmonotonic Dependency Net Update." LITH-MAT-R-82-23. Department of Computer and Information Science, Linkoping University, Linkoping, Sweden, 1982.

[Goodwin, 1984]
Goodwin, J. "WATSON: A Dependency-Directed Inference System." in *Nonmonotonic Reasoning Workshop*, Menlo Park, CA: AAAI, 1984, 103-114.

[Hogge, 1988]
Hogge, J.C. "Prevention Techniques for a Temporal Planner." *Proceedings AAAI-88*, Palo Alto, CA: Morgan Kaufmann, 1988, 43-48.

[Huff, 1981]
Huff, K.E. "A Database Model for Effective Configuration Management." *Proceedings Fifth International Conference on Software Engineering*, Washington, D.C.: IEEE Computer Society Press, 1981, 54-61.

[Huff & Lesser, 1982]
Huff, K.E. and Lesser, V.R. "Knowledge-based Command Understanding." Technical Report 82-6, Department of Computer and Information Sciences, University of Massachusetts, Amherst, MA, 1982.

[Huff & Lesser, 1987]
Huff, K.E. and V.R. Lesser, "The GRAPPLE Plan Formalism", Technical Report 87-08, Department of Computer and Information Sciences, University of Massachusetts, Amherst, MA, 1987.

[Huff & Lesser, 1987]
Huff, K.E., and Lesser, V.R. "A Plan-based Intelligent Assistant that Supports the Software Development Process." *Proceedings of Software Engineering Symposium on Software Development Environments,* New York: ACM Press, 1988, 97-106.

[Johnson, 1988]
Johnson, W.L. "Deriving Specifications from Requirements." *Proceedings Tenth International Conference on Software Engineering*, Washington, D.C.: IEEE Computer Society Press, 1988, 428-438.

[Johnson & Soloway, 1985]
Johnson, W. and Soloway, E. "PROUST: Knowledge-Based Program Understanding." *IEEE Transactions on Software Engineering*, 11:3 (March 1985), 267-275.

[Kaiser & Feiler, 1987]
Kaiser, G.E. and Feiler, P.H. "An Architecture for Intelligent Assistance in Software Development", *Proceedings Ninth International Conference on Software Engineering*, Washington, D.C.: IEEE Computer Society Press, 1987, 180-188.

[Kautz & Allen, 1986]
Kautz, H. and Allen, J. "Generalized Plan Recognition." *Proceedings AAAI-86*, Palo Alto, CA: Morgan Kaufmann, 1986, 32-37.

[Leblang & Chase, 1984]
Leblang, D.B. and Chase, R.P. "Computer-aided Software Engineering in a Distributed Workstation Environment." *Proceedings of Software Engineering Symposium on Software Development Environments*, New York: ACM Press, 1984, 104-112.

[Lefkowitz, 1987]
Lefkowitz, L.S. *Knowledge Acquisition through Anticipation of Modifications*. Ph.D. dissertation, University of Massachusetts, Amherst, MA, 1987.

[Lefkowitz & Croft, 1989]
Lefkowitz, L.S. and Croft. W.B. "Using a Planner to Support Cooperative Work." *Fifth IEEE Conference on AI Applications*, Washington, D.C.: IEEE Computer Society Press, 1989.

[Lesser, et al., 1988]
Lesser, V.R.; Pavlin, J.; and Durfee, E. "Approximate Processing in Real-time Problem Solving." *AI Magazine*, 9:1 (Spring 1988), 49-61.

[Litman, 1985]
Litman, D.J. *Plan Recognition and Discourse Analysis: An Integrated Approach for Understanding Dialogues*. Ph.D. dissertation, University of Rochester, Rochester, NY, 1985.

[Lukaszewicz, 1985]
Lukaszewicz, W. "Two Results on Default Logic." *Proceedings IJCAI-85*, Palo Alto, CA: Morgan Kaufmann, 1985, 459-461.

[Luria, 1987]
Luria, M. "Goal Conflict Concerns." *Proceedings IJCAI-87*, Palo Alto, CA: Morgan Kaufmann, 1987.

[Mahling & Croft, 1988]
Mahling, D.E. and Croft, W.B. "An Interface for the Acquisition and Display of Plans." Technical Report, Computer and Information Science Department, University of Massachusetts, Amherst, MA, 1988.

[McCarthy & Hayes, 1969]
McCarthy, J. and Hayes, P.J. "Some Philosophical Problems From the Standpoint of Artificial Intelligence." *Machine Intelligence 4*. Edited by G. Meltzer and D. Michie. Edinburgh: Edinburgh University Press, 1969.

[McDermott, 1978]
McDermott, D. "Planning and Acting." *Cognitive Science*, 2 (1978), 71-109.

[Morris, 1987]
Morris, P. "Curing Anomalous Extensions." *Proceedings AAAI-87*, Palo Alto, CA: Morgan Kaufmann, 1987, 437-442.

[Osterweil, 1987]
Osterweil, L. "Software Processes are Software Too." *Proceedings Ninth International Conference on Software Engineering*, Washington, D.C.: IEEE Computer Society Press, 1987, 2-13.

[Parnas, 1972]
Parnas, D. "On the Criteria to be Used in Decomposing Systems into Modules." *Communications of the ACM*, 15:3 (March 1972).

[Pednault, 1988]
Pednault, E.P.D. "Extending Conventional Planning Techniques to Handle Actions with Context-dependent Effects." *Proceedings AAAI-88*, Palo Alto, CA: Morgan Kaufmann, 1988, 55-59.

[Penedo & Stuckle, 1985]
Penedo, M.H. and Stuckle, E.D. "PMDB—A Project Master Database for Software Engineering Environments." *Proceedings Eighth International Conference on Software Engineering*, Washington, D.C.: IEEE Computer Society Press, 1985, 150-157.

[Petrie, 1987]
Petrie, C. J. "Revised Dependency-Directed Backtracking for Default Reasoning." *Proceedings AAAI-87*, Palo Alto, CA: Morgan Kaufmann, 1987, 167-172.

[Potts, 1984]
Potts, C., ed. *Software Process Workshop '84*, Washington, D.C.: IEEE Computer Society Press, 1984.

[Ramamoorthy et al., 1985]
Ramamoorthy, C.V.; Usuda,Y.; Tsai, W.T.; and Prakash, A. "GENESIS: An Integrated Environment for Supporting Development and Evolution of Software." *Proceedings Ninth International Computer Software and Applications Conference*, Washington, D.C.: IEEE Computer Society Press, 1985, 472-479.

[Reiter, 1978]
Reiter, R. "On Closed World Databases." in *Logic and Databases*. Edited by H. Gallaire and J. Minker. New York: Plenum Press, 1978, 55-76.

[Reiter, 1980]
Reiter, R. "A Logic for Default Reasoning." *Artificial Intelligence*, 13 (1980), 81-132.

[Reiter & Criscuolo, 1981]
Reiter, R., and Criscuolo, G. "On Interacting Defaults." *Proceedings IJCAI-81*, Palo Alto, CA: Morgan Kaufmann, 1981, 270-276.

[Rich, 1981]
Rich, C. "A Formal Representation for Plans in the Programmer's Apprentice." *Proceedings IJCAI-81*, Palo Alto, CA: Morgan Kaufmann, 1981, 1044-1052.

[Rich, 1985]
Rich, C. "The Layered Architecture of a System for Reasoning About Programs." *Proceedings IJCAI-85*, Palo Alto, CA: Morgan Kaufmann, 1985, 540-546.

[Rich, 1986]
Rich, C. "The Essential Monotonic and Nonmonotonic Truth Maintenance Systems." personal communication, 1986.

[Rich & Shrobe, 1978]
Rich, C. and Shrobe, H.E. "Initial Report on a Lisp Programmer's Apprentice." *IEEE Transactions on Software Engineering*, 4 (November 1978), 456-467.

[Rich & Waters, 1988]
Rich, C. and Waters, R.C. "The Programmer's Apprentice: A Research Overview." *IEEE Computer*, November 1988, 10-25.

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

[Sidner & Israel, 1981]
Sidner, C. and Israel, D. "Recognizing Intended Meaning and Speaker's Plans." *Proceedings IJCAI-81*, Palo Alto, CA: Morgan Kaufmann, 1981, 203-208.

[Smith et al., 1985]
Smith, D.R.; Kotik, G.B.; and Westfold, S.J. "Research on Knowledge-based Environments at Kestrel Institute." *IEEE Transactions on Software Engineering*, 11:11 (November 1985), 1278-1295.

[Stallman & Sussman, 1977]
Stallman, R.M., and Sussman, G.J. "Forward Reasoning and Dependency-directed Backtracking in a System for Computer-aided Circuit Analysis." *Artificial Intelligence*, 9 (1977), 135-196.

[Stefik, 1981a]
Stefik, M. "Planning with Constraints." *Artificial Intelligence*, 16 (1981), 111-140.

[Stefik, 1981b]
Stefik, M., "Planning and Metaplanning." *Artificial Intelligence*, 16 (1981), 141-169.

[Sullivan & Cohen, 1985]
Sullivan, M. and Cohen, P.R. "An Endorsement-Based Plan Recognition Program." *Proceedings IJCAI-85*, Palo Alto, CA: Morgan Kaufmann, 1985, 475-479.

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

[Tichy, 1985]
Tichy, W.F. "RCS—A System for Version Control." *Software Practice and Experience*, 15:7 (July 1985), 637-654.

[Tully, 1989]
Tully, C.J., ed. "Proceedings of the Fourth International Software Process Workshop." *Software Engineering Notes*, 14:4 (June 1989), 1-174.

[Vere, 1983]
Vere, S.A. "Planning in Time: Windows and Durations for Activities and Goals." *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 5:3 (May 1983), 246-267.

[Waters, 1985]
Waters, R.C. "The Programmer's Apprentice: A Session with KBEmacs." *IEEE Transactions on Software Engineering*, 11:11 (November 1985), 1296-1320.

[Wile & Allard, 1986]
Wile, D.S. and Allard, D.G. "Worlds: an Organizing Structure for Object-Bases." *Proceedings of Software Engineering Symposium on Software Development Environments,* New York: ACM Press, 1986, 16-26.

[Wileden & Dowson, 1986]
Wileden, J.C. and Dowson, M., eds. "Proceedings of the Second International Software Process Workshop." *Software Engineering Notes*, 11:4 (August 1986), 1-74.

[Wilensky, 1981]
Wilensky, R., "Metaplanning: Representing and Using Knowledge About Planning in Problem Solving and Natural Language Understanding." *Cognitive Science*, 5 (1981), 197-233.

[Wilensky, 1983]
Wilensky, R. *Planning and Understanding*. Reading, MA: Addison-Wesley, 1983.

[Wilensky et al., 1984]
Wilensky, R.; Arens, Y.; and Chin, D. "Talking to UNIX in English: An Overview of UC." *Communications of the ACM*, 27:6 (June 1984), 574-593.

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

[Wilkins, 1985]
Wilkins, D.E., "Recovering From Execution Errors in SIPE." TN 346, SRI International, Menlo Park, CA, 1985.

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

[Williams, 1988]
Williams, L. "Software Process Modeling: A Behavioral Approach." *Proceedings Tenth International Conference on Software Engineering*, Washington, D.C.: IEEE Computer Society Press, 1988, 174-186.

[Wills, 1986]
Wills, L.M. "Automated Program Recognition." Tech. Report 904 (M.S. thesis), Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA, 1986.