University Of Alberta

Clarification Dialogues for Plan Recognition in Advice-Giving Settings

by

© Kenneth J. Schmidt

A thesis
submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of
Master of Science

Department Of Computing Science

Edmonton, Alberta
Spring, 1994

ISBN 0-612-11358-2

Canada

UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: Kenneth J. Schmidt

TITLE OF THESIS: Clarification Dialogues for Plan Recognition in Advice-Giving Settings

DEGREE: Master of Science

YEAR THIS DEGREE GRANTED: 1994

Kenneth J. Schmidt

Box 1710
Westlock, Alberta, Canada
T0G 2L0

Date: April 20, 1994

UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Clarification Dialogues for Plan Recognition in Advice-Giving Settings** submitted by **Kenneth J. Schmidt** in partial fulfillment of the requirements for the degree of **Master of Science**.

Dr. Peter van Beek

Dr. John Buchanan

Dr. Xiaoling Sun

Dr. Robin Cohen
External

Date: April 19, 1994

Abstract

Advice-giving systems and question-answering systems may employ plan recognition to provide more cooperative responses to user queries. A major problem of such systems will occur when a user's actions are consistent with more than one plan. One approach to the problem of ambiguous plans has focused upon the [redacted] determination of a single plan to explain the user's actions. This approach handles the danger of committing to the wrong plan. Another approach is that of clarifying [redacted] with the user. The system queries the user about the possible plans, in order to [redacted] ambiguity. Such clarification dialogues are not problematic for small domains. [redacted] we allow that any complex domain, deemed suitable to warrant a practical [redacted] system, may involve numerous detailed and complex plans. In such advice-giving settings, it is desirable to reduce, and attempt to minimize, the clarification dialogue required to resolve ambiguity. Little research has been done in this area of clarification of detailed and complex plans. We propose several strategies and methods aimed at reducing clarification dialogue in advice-giving settings. These strategies enhance and build upon previous work in this area that provides a basic approach to the problem. We have combined these methods into a new general clarification framework that allows the handling of complex and detailed plans. Our clarification process never overcommits in recognizing the user's plan and never backtracks into a debugging dialogue with the user. It provides a formulation of *what* to ask the user and identifies situations in which clarification dialogue may be further reduced, in order to more quickly respond to a user's query. In particular, we have identified situations in which unnecessary clarification can be avoided by skipping questions about irrelevant parts of plans and focusing on key actions that identify critical sets of plans that matter to the ambiguity.

Acknowledgments

I would like to especially thank my supervisor, Peter van Beek, for his invaluable help and input toward this research. I would also like to extend a special thank you to both Peter van Beek and Robin Cohen for allowing me the opportunity of co-authoring related papers on the subject material. I also thank the Natural Sciences and Engineering Research Council of Canada for their financial support. Lastly, I would like to thank my wife, Joanne, as well as our children: Mark, Allison, Heather and Cathy, for their support and perseverance.

Table of Contents

List of Figures

Chapter 1. Introduction

As research advances in the field of Artificial Intelligence, the users of interactive automated consultants or advice-giving systems will expect these systems to approach the level of competence and cooperation received from human experts. When we consult with human experts, we expect much more than simple 'yes' or 'no' replies to answer our questions. We expect human experts to provide cooperative responses that provide additional information or advice to help in achieving our goals, prevent us from being misled, or at least warn us if our intended actions or goals may be faulty. In order to provide such cooperative responses, the human expert must be able to determine a client's plans or intentions, often from interactive dialogue with the client. As an example of this, consider the following hypothetical conversation between a new university student and a general course-advising councilor:

Student: I'm interested in learning about using computers for music applications. Can I take the computer course, CMPUT 199, on advanced programming languages?

Advisor: Do you intend to obtain a Bachelor of Arts degree?

Student: Yes.

Advisor: The quota for this course has not yet been filled, so it is possible for you take the course as a science option. However, this particular course involves intensive programming and is generally not recommended for first year Arts students. See your particular department advisor if you wish to take the course. Recommended computer courses are CMPUT 161 or CMPUT 172 on computer applications.

This conversation illustrates several points. The student has not yet registered at the university, so the councilor has no record or background knowledge about the student. It was possible for the student to be involved in numerous different programs that the university had to offer. The advisor did not really ascertain the exact program (or plan) of the student. For example, the advisor did not establish which Arts department the student was intending to enter, nor whether the student intended upon an honors or specialization program. This may be established as the dialogue continues. However, the advisor did clarify the faculty the student intended to enroll in, by asking the student. Based on this, the advisor provided a cooperative response to the student's question by explaining why the course was not suitable and offering alternatives. By asking one *clarification* question, the advisor was able to provide an appropriate response to the query.

Several issues are involved in the automation of 'the councilor' into an interactive advice-giving system. Some of these are natural language generation issues, plan recognition, principles of cooperative behaviour, databases of domain knowledge and user interface design. In this thesis, the primary focus is that of the clarification aspect of such automated systems. The system must determine *when* it is necessary to clarify ambiguous plans with the user. The system must also determine *what* to ask the user in such clarification dialogues, in order to reduce the dialogue length. When the possible plans for an advice-giving session increase in number or complexity, this aspect becomes increasingly important. The problem, then, is that of reducing clarification dialogues for plan recognition (and cooperative response generation) in advice-giving settings.

1.1. Clarification Dialogue

Several researchers have pointed out the importance of some form of plan recognition, in automated systems, for the purpose of providing cooperative responses to the users of such systems. Plan recognition is a process that identifies possible plans from the observation of an agent's actions (or a description of actions). Once a user's plan or intention is recognized, a cooperative response can be provided for user queries based upon the plan and its suitability to the user's intended goal. A major difficulty occurs when the user's action or query is consistent with more than one plan. The ambiguity of the plans may affect which response is provided. Heuristics may be used to commit to a single plan, but this approach involves several problems that occur when the selected plan is incorrect. Another approach, for interactive systems, is that of asking the user certain questions in order to clarify which plans are being pursued. This approach avoids the issue of overcommitting to the wrong plan and backtracking to determine the correct plan. However, if we are permitted the use of such clarification dialogues, we do not want the dialogues to be lengthy, verbose, or confusing to the user. Such dialogues will only serve to make the system appear to be unintelligent, awkward, and even uncooperative.

This thesis presents a general framework for clarification dialogues. Several strategies, methods, and guidelines are incorporated, all with the perspective of reducing or minimizing clarification dialogue, but retaining coherence, to provide advice-giving sessions agreeable to the user. We build upon previous research in this area, particularly the work of van Beek and Cohen [van Beek and Cohen 91; van Beek, et. al. 93] on resolving plan ambiguity for cooperative response generation. In their work, van Beek and Cohen address the problem of what the system should do when the user's actions are ambiguous and consistent with more than one plan. The key contributions of their work are in providing a clear criterion for when to respond to a question with a question that will differentiate ambiguous plans, and a proposal of a specific solution to what questions should be asked of the user. Their solution employs the use of two key features. The first is that of *critiquing* possible plans and identifying plans with the same *fault*, for the purpose of determining when the system should respond. For example, a plan to entertain a guest by serving a meat dish meal would be a faulty plan if the guest is a vegetarian. Critiquing such a plan would annotate the plan with a fault concerning the vegetarian guest. A response is given if all possible plans have the same fault annotation; if not, the system enters into a clarification dialogue. The second key feature is that the plans have a *hierarchical* structure represented by an *event hierarchy* that is similar to the plan representation of Kautz [Kautz 87]. This hierarchy of events (or actions) is used in determining what to ask the user, by employing a top-down traversal of the hierarchy. This thesis expands upon this approach, which we term to be the *basic approach*. We build upon this basic approach by adding new strategies, and addressing some of its limitations.

Given a set of ambiguous plans, the purpose of clarification dialogue is that of reducing the ambiguity, by pruning the possible plans, to the point at which such ambiguity no longer matters for the purpose of providing a response. The pruning is not necessarily to a single plan. We want to minimize, or reduce as much as possible, this clarification dialogue. To achieve this, we basically develop strategies and methods that attempt to reduce the number of clarification questions that the system asks of the user. It is also important to maintain a coherency to clarification questioning and to keep such questions from becoming overly lengthy or complex. Our strategies also consider this aspect.

1.2. Summary of Key Contributions

The general clarification framework that we propose is capable of handling *complex* plans that involve multiple necessary steps (i.e., necessary sub-plans). The basic approach could not adequately deal with such complex plans. It was basically limited to single observations from which hierarchies of non-complex plans would result from plan recognition. Since our framework is capable of dealing with complex plans, it allows the possibility of multiple observations that may often be related to different sub-branches of those plans. This capability is achieved by maintaining the event hierarchy in a tree-like structure and specifying a stack data structure to keep track of multiple sub-branches.

In reducing the number of clarification questions asked, the problem is basically that of choosing the right question, or the best question, in order to reduce ambiguity as quickly as possible. The basic approach employs a critiquing component that assigns fault annotations (a set of faults) to recognized plans. To provide a response, clarification continues, eliminating possible plans, until all remaining plans have the same fault annotation. At this point ambiguity no longer matters to providing a response to a user query. To determine if ambiguity matters, the plans are partitioned according to their fault annotations. All plans of a particular partition will have the same fault annotation (i.e., they all have the same set of faults). Ambiguity does not matter if there is only one partition. Ambiguity matters if there is more than one partition. Here, we propose maintaining these *fault partition* data structures and exploiting them to select clarifying events to ask the user. Most of the following methods and strategies utilize this fault partitioning of the possible plans.

At any disjunctive branch point in the clarification process, there will be a set of alternative actions that indicate alternative possible plans. The basic approach clarifies these alternatives by asking a single 'Yes/No' type question about a single alternative, or by asking a single menu-type question in which each menu item is an alternative action of the branch point. We also adopt this questioning format for the purpose of clarity to the user and providing a basic platform for clarification dialogues. Should menu-type questions not be used, the basic approach allowed the use of likelihood factors in selecting which alternative to ask first. In the absence of statistical information to determine one plan over another, the selection of the alternative for the next 'Yes/No' question was left underspecified. Here we propose a default criterion of rules to determine the next alternative to ask, in the case that menus are infeasible and information on the likelihood of plans is not available. We determine the number of remaining fault partitions to both a 'Yes' and 'No' reply for each alternative. We select the alternative to ask next, based upon the results. The rules are designed to eliminate partitions as rapidly as possible until ambiguity no longer matters. Our general framework also allows the use of menus and likelihood factors, providing an initial toggle between these methods, as well as the default situation.

Plans that involve several levels of abstraction and complex plans with multiple sub-branches are represented by event hierarchies that have several disjunctive branch points. In certain situations, plans will diverge following different alternative actions, but converge again to some common action or to some necessary sub-branch. These situations allow the possibility of skipping branch points for clarification questioning, while maintaining a top-down traversal of the event hierarchy. We propose a criterion as to when the clarification process should explore the possibility of skipping branch points as well as a method to determine when branch points should indeed be skipped. If all of the alternatives of a branch point are represented in every fault partition, then it may be possible to skip that particular branch point. We first determine some new branch point to

skip to, and then establish sets of events that uniquely identify the plans of each fault partition. These sets allow us to determine if the ambiguity can be resolved from the new branch point. If the remaining partitions can be reduced to one partition by only asking about events occurring after the branch point that is skipped to, then we confirm that it is safe to skip ahead to that branch point.

We have identified certain situations in which we are able to abandon the top-down traversal of the event hierarchy, in order to effectively reduce clarification dialogue. When all plans of a particular fault partition share a specific event that is not held by other plans, that event uniquely identifies the plans of that partition. We term such events to be *key events*. By asking the user a 'Yes/No' question about pursuing a key event, we can substantiate or eliminate an entire partition. For example, a 'Yes' reply to a key event will substantiate all plans of a single partition. Since no other plans hold this event, they are eliminated from further consideration. The remaining plans will all belong to one partition and their ambiguity will not matter to providing a response. The use of key events may jeopardize dialogue coherency. For this reason, our present clarification framework limits their use. However, we briefly describe an extension of the concept to events that uniquely identify all plans of more than one partition.

We introduce the notion of *overriding* faults for the purpose of avoiding pointless clarification. If clarification dialogue determines that the user is intending to pursue an action that is inappropriate for an observation, the user should be warned directly. In such a case, we propose that clarification terminate prematurely, even though unresolved plan ambiguity still matters to providing a more precise response. Overriding faults override, or take precedence over, other faults of a plan. They indicate that a certain action should not be pursued for a particular observation. Such faults are associated with these actions. If we determine that a user intends to pursue such an action, we terminate further clarification of any specializations of that action, and provide a response that addresses the overriding fault of the action.

All of the above strategies and methods are incorporated and unified into a single general clarification framework or process, for soliciting clarification from users for plan recognition. This is not meant to imply that the framework must be taken as a whole, but illustrates how the different strategies can be integrated. The framework considers both menu-type clarifying questions as well as 'Yes/No' questions. It allows the use of different plan likelihoods as well as providing a basic default approach in the absence of any likelihood factors. We have provided general algorithms and examples for the procedures involved in the framework as well as basic data requirements, to bring the framework up to the stage of implementation and empirical testing.

1.3. Thesis Organization and Outline

Chapter 2 of this thesis provides necessary background information. A general historical background and setting is given for advice-giving systems that deal with plan recognition for the purpose of cooperative response generation. Also, some related work in clarification dialogue is discussed, but there is relatively little research in the area of reducing such dialogue in the presence of detailed or complex plans. That is the primary topic addressed by this thesis.

Chapter 3 provides a setting and detailed description of the basic approach that we build upon, as well as outlining its limitations. The chapter also gives a focused terminology for terms that are often repeated. Throughout this thesis, two domains are used for examples and illustration, which are introduced in this chapter. The cooking domain is quite small and is used primarily to clearly illustrate the key contributions of this thesis. The course-advisor domain provides a more practical setting and a larger scale of plans.

Chapter 4 presents the general clarification process in a unified framework. The different methods and strategies are presented incrementally, with examples of their use. The algorithms are presented from top-level procedures toward more lower-level procedures. The chapter also includes a brief discussion about different questioning and response formats.

Chapter 5 provides a brief summary and discussion of open problems and future research directions.

Chapter 2. Background

Although plan recognition and cooperative response generation can be viewed as two separate areas of Artificial Intelligence research in their own right, the focus here is upon their useful combination in question-answering or advice-giving systems; and more specifically, upon those systems that use some type of interactive clarification process.

Plan recognition is a process that infers or constructs an agent's plan or plans from observation of the agent's actions. In our case, descriptions of actions serve as observations that describe past or present actions, or the intention to do certain future actions. Plan recognition, by itself, is useful in many areas, such as story understanding, discourse modeling, modeling naive psychology, intelligent computer system interfaces, and strategic planning (see citations in [Kautz 87]). Our interest in plan recognition lies in its usefulness in question-answering or advice-giving systems, where recognizing the particular plans underlying a user's queries is an invaluable aid to generating appropriate and cooperative responses to those queries.

Cooperative response generation, by itself, does not necessarily go hand in hand with plan recognition. That is, it is not necessary to recognize a user's plan in order to provide a *more* cooperative response to a user's query. For example, consider the following dialogue between a questioner (Q) and respondent (R) [Webber 87]:

> Q. Which ships have a doctor on board?
> R. The JFK, the Admiral Nimitz . . . .

The questioner concludes that these ships have a doctor and all others do not. More precisely, the questioner is led to believe in the existence of a non-empty set of ships without doctors. However, if the listed ships are the only ships, the response is misleading, even though it is correct. A better response might be "All 43 of them". This can be seen as a more cooperative or helpful response in that it prevents the user from being misled. Plan recognition was not involved in providing this more cooperative response. Webber points out that for an answer to be cooperative, it must be correct, useful, and not be misleading. Plan recognition is of prime importance in enabling more useful answers to be provided to an agent. The following simple example [Allen 83] illustrates this point:

> An agent (A) is carrying an empty gas can and comes up to a person on the street (S), and asks '*Where is the nearest gas station?* '; to which S answers '*On the next corner* ', knowing full well that the station is closed.

Although the answer given by S is correct and not really misleading, it is of little use to A, and can be viewed as being uncooperative on the part of S, if we assume that S correctly recognized A's plan (i.e., to go to the nearest gas station in order to get gasoline for the empty gas can). A more cooperative response might have been

> *'On the next corner, but it's closed. There is an open gas station about three blocks away in the other direction '.*

This response not only provides the correct answer to the question (i.e., 'On the next corner'), but is cooperative, in that it warns of an obstacle in the agent's plan (i.e., the nearest gas station is closed) and also suggests a better alternative plan to achieve the

agent's goal. Our particular interest, here, is with systems in which plan recognition plays a central role in cooperating with an agent, to help that agent in achieving a goal or warning the agent about faulty plans to achieve a goal. As such, these types of systems fit more into the area of advice-giving systems, where the primary concern is in helping the user with respect to the user's higher-level goals. In the future, such automated consulting systems will have application in many diverse domains. Some of the many possibilities are:

- financial advisors
- career consulting systems
- course advisor and schedule-planning systems
- telephone registration systems or automated telephone consultants
- automated tutorial/training or help systems for complex domain environments
- complex maintenance systems
- holiday planning and travel agency systems

For any system, gathering the exact content of a cooperative response to a query and presenting the answer to the user in a natural language form is an important issue. Our concern is not really in this area, and we refer the reader to recent work by Gaasterland on generating cooperative answers in deductive databases [Gaasterland 92]. The advice-giving type settings that we envision here, require plan recognition to be of primary importance in selecting responses that are cooperative and aid users in achieving their goals. The following section provides an historical framework of previous work on plan recognition for cooperative response generation. Clarification is the process that deals with resolving ambiguity of recognized plans, interactively with the user, in order to provide a cooperative response. To our knowledge, very little research has been done in this area and clarification is the problem to which this thesis is addressed.

2.1. Plan Recognition and Cooperative Response Generation

2.1.1. Genesereth's ADVISOR

The early work by Allen and Perrault on recognizing and analyzing intentions from speech utterances [Allen 83; Allen and Perrault 80] is viewed as a foundation to much of the work in the area of plan recognition and cooperative response generation. Before describing their contribution, the work done by Genesereth, with respect to the role of plans in automated consultation [Genesereth 79], should be discussed, as it raises several general issues related to advice-giving systems.

Genesereth developed one of the earliest automated consultants, called the ADVISOR. Its purpose is that of helping novice users with a powerful computer program for symbolic mathematics (called MACSYMA). Upon receiving an erroneous result to a series of MACSYMA commands, the user can ask the ADVISOR for help and receive advice about inappropriate commands, as well as suggestions of commands that should work.

The ADVISOR's implementation is based on the assumption that novice users behave in accordance with a standard heuristic problem solving algorithm, called MUSER. In solving a problem, the user is assumed to implicitly generate a goal-subgoal graph called a plan. To the user, this plan represents a direct proof that the commands used achieve the user's goal (i.e., the user believes the plan is correct). The consultation process of the ADVISOR first involves reconstructing the user's plan, through a combined process of dialogue and automatic plan recognition. Once a complete plan is

obtained, its underlying beliefs are checked for errors, that can then be reported to the user.

The data made available to the ADVISOR is the relevant sequence of MACSYMA commands used and the overall goal of the user. If this goal is not supplied, the ADVISOR obtains it by asking the user for a statement of the goal. The sequence of MACSYMA commands can be viewed as an observation, by the system, of multiple actions performed by the user (*later, we deal with certain problematic aspects of such multiple observations*). These commands are assumed to represent the fringe of the user's plan, with the goal being the root. The plan reconstruction procedure is used to fill in the intervening structure. The first part of this process involves the conversion of the user's actions into a dataflow graph. The second step is mechanical plan recognition involving a heuristic parsing to search the partially reconstructed plan for connecting plan fragments from a plan Library or error fragments from an error library. The ADVISOR also employs the MUSER model as data in inferring further plan structure. The process continues until a complete plan is found that ties the user's goal to the graph. The last phase in obtaining the user's plan is termed as user interrogation. Should the ADVISOR's plan recognition procedure fail in producing a complete plan, the ADVISOR tells the user what it has figured out and asks the user for help. Once the user's plan is established, the ADVISOR attempts to identify a misconception, either by recognizing some standard error in the user's plan or by a general debugging process. The ADVISOR confirms a suspect misconception by asking the user whether or not it is believed. Also, the ADVISOR has the capability of executing alternative commands from its own data base, and if successful, informs the user.

Apart from the implementation itself, Genesereth points out several issues of interest to the area of automated consultation. Some of these issues are the following:

- The need for consultation in any sufficiently complex domain in which a user has a problem solving situation, but lacks the knowledge to fully understand the domain. This may be incidental, where the domain is simple, but time constraints do not allow the user to learn the necessary knowledge. On the other hand, it may be essential, for the domain may be too complex for the user to learn everything.
- A good consultant should not only possess a substantial knowledge of its material, but also a good model of the user's knowledge. Genesereth's key point is that plan reconstruction and analysis is a good way of gaining such a model.
- The notion that a person's plan is central to consultation and should prove useful for such types of man-machine interaction in general.
- The use of both correct plans as well as incorrect and buggy plans.

If we accept that plan recognition does play a vital role in automated consultation, then an obvious question is 'why not just ask the user to describe their plan in detail?' Genesereth points out that human consultants are able to bypass most and all such questioning. It would be desirable to have automated consultants do the same. Genesereth also points out that in practice, users often will not know what terms to use in describing their intentions, or the description may be too tedious for their patience to bear. To be effective, Genesereth suggests that a consultant must reconstruct as much of the user's plan as possible, with as little questioning as possible.

An important point to note is the ADVISOR's use of interactive clarifying dialogue in deriving the user's plan. This dialogue is used to obtain the user's goal, fill in missing pieces of a plan, and to confirm a tentative plan (the user's misconception in a faulty plan).

Genesereth did not formalize many of the principles behind the ADVISOR, except in the form of a LISP implementation. Issues of multiple goals and concurrent multiple plans were not dealt with, and these play an important role in other plan recognition systems such as that of Kautz [Kautz 87].

2.1.2. Intentions from Utterances

The work of Allen and Perrault [Allen 83; Allen and Perrault 80] on recognizing and analyzing intentions from natural language utterances is primarily aimed at providing a plan-based model to explain a wide range of linguistic behaviour and which would be useful for the design of question-answering systems. In particular, they show how their model can account for responses that provide more information than that which is explicitly requested, as well as responses to incomplete sentence fragments and indirect speech acts. The underlying basis for this lies in the ability to infer the speaker's plans and detect obstacles in those plans. The concepts of recognizing plans and generating helpful responses are combined in a formal manner. They implement the model on a simple train station domain to show that such work can lead to more helpful, but still practical, natural language understanding systems.

As an example of providing more information than that requested, Allen and Perrault give the following exchange between a patron and clerk at an information booth in a train station:

> **patron:** When does the Montreal train leave?
> **clerk:** 3:15 at gate 7.

Allen and Perrault's model explains the clerk's response as being motivated by the clerk wanting to assist the patron in furthering the patron's goals. The response not only contains the train departure time, which was explicitly requested, but also the departure location. From the question, the clerk was able to infer the patron's plausible plan of boarding the train to Montreal. Obstacles to executing this plan were the patron's lack of knowledge of the departure time and the departure location. In order to be more helpful and cooperative, the clerk's response also included the departure location.

The plan recognition suggested by Allen and Perrault involves a set of plan recognition rules, to find plausible inferences, along with a heuristic control strategy . The rules have the form:

> If the hearer believes that the speaker wants proposition P
> to hold,
> then the hearer believes that the speaker wants action A,
> given that P is a precondition (or effect) of A.

The rules are applied to form chains of inference (or partial plans) which attempt to connect a single observed action called an *alternative*, to a domain-dependent goal called an *expectation*. These chains are expanded in a best-first manner. Each has a numerical score from (a somewhat confusing) heuristic rating system, intended to find the most likely plan. The process terminates when a complete plan (connecting alternative and expectation) has a significantly higher rating than any other chain. Obstacles to the execution of this plan can then be detected and included in a helpful response to the speaker.

Allen and Perrault's model also addresses clarification dialogues, but to a limited extent. Their example is the following:

Patron: When is the Windsor train?
Clerk: To Windsor?
Patron: Yes.
Clerk: 3:15.

From the patron's question, two plans are possible. The patron either wants to meet the train from Windsor or board the train going to Windsor. There is no reason to accept one plan over the other. Allen's model accepts both plans and detects an ambiguity. The expectation that first occurred in the search is then asked about. The answer asserts only one plan to be accepted, after which obstacle detection and response continue as usual.

It was not Allen's intent to deal extensively with clarification, but rather to explain it through the model. However, several issues should be pointed out with respect to this clarification. First, Allen's train station domain is not very complex, having basically only two expectations or goals; that of boarding a train or meeting a train. Also, there are no real alternative methods involved in achieving these goals, so that the goal is actually equivalent to the plan in terms of describing the plan. In this case, clarification is a simple matter, distinguishing between only two plans, both of which can be concisely and uniquely described by reference to their top-level goals. The clarification process is not so simple when the plans become more complex. Several top-level goals may equally apply to an observation. Each of these top-level goals may have alternative actions or ways of achieving them, and each of these actions, in turn, may also have alternative methods of achievement, and so on. It may not be possible to uniquely distinguish a plan by describing the top-level goal, nor may it even be possible through reference to any single action of the plan. For example, we cannot adequately describe the route (plan) of a long vacation trip by reference to a single intervening city.

Allen's system did not deal with multiple observations, and the domain is not complex. However it established a basic framework, in the area of plan recognition and cooperative response generation, which other researchers extended.

2.1.3. Plans and Extended Dialogue

Sidner and Israel [Sidner and Israel 81; Sidner 83] presented a model of interpretation of speaker meaning which built upon Allen's framework. This model takes into account several different kinds of belief about the current situation of the speaker, the speaker's goals, the discourse context, and the speaker's knowledge of the hearer's capacities and conventions for acting. The model produces an explanation for these beliefs that is then used to provide responses, and to draw further conclusions about the speaker's intentions. An important objective of their model is the recognition of the speaker's intended meaning as opposed to the semantic meaning (e.g., the statement of "You're a Prince" has a different meaning depending on the context and situation in which it is used) The intended meaning is relevant for communicative situations such as natural language systems. In their model, declaratives such as statements of desires or complaints can be used in recognizing a speaker's plans, as well as commands or interrogative utterances. Sidner expands upon Allen's model by recognizing a somewhat richer form of plans (plans that have multiple steps) and making an explicit connection between the speaker's intentions and the speaker's intended response. That is, the speaker intends to get a certain response from a certain declarative statement, via the hearer's ability to recognize the speaker's plans and intentions.

Carberry proposed a model for tracking the changing task-level goals of a speaker during extended discourse in an information-seeking environment [Carberry 83]. This model allows a complex domain of goals and plans in which the user's plan is incrementally built as dialogue progresses. It maintains two different forms of context processing: a local plan (or goal) context as well as a global plan context. These can be used to differentiate past goals from current goals, or to incorporate previous plans into an overall plan context. The plans in Carberry's system, called TRACK, are hierarchical structures of goals and actions. Each of these goals and actions has an associated plan, or represents a primitive in the domain. A STRIPS formalism [Fikes and Nilsson 71] is used to represent the plans, so that a plan has preconditions, a set of partially ordered actions, and a set of effects. The plans can expand to different levels of detail, containing the goals and actions of other fully expanded plans. The mechanism included plan recognition rules of Allen, Sidner and Israel. Carberry points out that in more complex information-seeking environments, the user may investigate several low-level subgoals, that may be components of many different higher-level plans, and then relate them as parts of a specific plan. A complete plan, in most cases, may not be immediately evident from one or two utterances in the first part of a dialogue. Carberry's system builds the plan as dialogue progresses, keeping track of the user's current focus within that plan, and thereby eliminating the need for working with many separate complete plans at once. Thus, Carberry's model accounts for multiple observations.

Carberry's model can request clarification from the user should more than one focused plan exist and there is uncertainty about the speaker's goals. This clarification is apparently done only if the ambiguity prevents a response to the utterance, or if it is believed that an obstacle exists that prevents the speaker from pursuing one of the candidate focused goals. The mechanism for this, however, is not elaborated upon nor specified. The model also employs controlling heuristics (but not those used by Allen) to determine candidate focused plans from derived goals, and to determine a focused plan from among the candidates. The underlying assumption was to obtain one unique plan to explain the observations.

The work of Joshi, Webber, and Weischedel on preventing false inferences [Joshi, Webber, and Weischedel, 1984; Joshi *et. al.*, 1984], broadened the scope of what was to be expected from responses in cooperative man-machine interaction. They maintain that such systems should modify a response, if there is reason to believe that the original response may mislead the user. They point out certain types of informing behaviour that should be expected from cooperative systems, with respect to the user's stated goal as well as the user's *intended* goal. The intended goal can be viewed as a higher-level goal of the user. For example, a user question of 'Can I drop course X ?' may have the intended goal of avoiding failure of the course. A simple 'yes' or 'no' answer that only addresses the stated goal of dropping the course is truthful and informative, but not sufficient. The user expects more from a cooperative agent. For instance, a 'no' response should contain an alternative course of action, or point out that there is no such alternative, providing the cooperative agent knows about such actions. Also, an explanation of the 'no' answer would be expected. Similarly, 'yes' answers should also point out better ways of accomplishing a higher-level goal if such other plans exist, and they should certainly contain a warning comment if the stated goal is possible, but there is some problem with the intended goal (e.g., 'Yes, you can drop the course, but your mark will still be recorded as a withdrawal while failing'). In order to provide such responses, a system must have some knowledge of the different actions that can lead to a goal, and must be able to recognize or infer an intended goal. Later in this dissertation, these different response types will be accounted for by *fault annotations* of plans. That is, better plans and alternative plans, are viewed as causing a flaw or fault with the particular plan being

considered, because it is sub-optimal in the case of better plans or cannot be achieved in the case of alternative plans.

A different model of plan inference in cooperative question-answering was proposed by Pollack, in which the beliefs of actors and observers are distinguished, and not assumed to be identical with respect to actions in the domain [Pollack, 1986]. Plans are viewed as mental phenomena (or a state of mind), and having a plan is analyzed as having a particular configuration of beliefs and intentions. Invalid plans can be associated with particular discrepancies between the beliefs ascribed to the actor (by the observer) and the beliefs actually held by the observer. Pollack claims that the content of a cooperative response to such invalid plans is affected by the type of discrepancy. One example (from Pollack's SPIRIT implementation in the domain of computer mail) involves the following responses to a question:

(Question) I want to prevent Tom from reading my mail file. How do I set the permissions on it to faculty-read only?

(Response1) Well, the command is SET PROTECTION = (Faculty:Read), but that won't keep Tom out: file permissions don't apply to the system manager.

(Response2) Well, the command is SET PROTECTION = (Faculty:Read), but that won't keep Tom out: he's the system manager.

From the user's question, a simple plan (i.e., preventing Tom from reading a file by changing the file permissions) is ascribed to the user, but it is judged to be ill-formed and invalid. The two different responses indicate different discrepancies of belief. In Response1, both the system and user believe Tom is the system manager; the discrepancy of belief concerns the effect of file permissions on system managers. In Response2, both the system and user believe file permissions do not affect system managers, but the discrepancy of belief is about who is the system manager. Pollack expands her model to distinguish between ill-formed plans and incoherent plans. In the former case, the system does not believe that a certain user-action will lead to another specific action, while in the latter case, the system cannot make any sense out of the query, and this should be revealed in the response.

Here, we do not examine such critical issues of erroneous or incoherent plans (i.e., plans that do not easily fit into the domain being considered), nor do we distinguish between user and system beliefs. Our plan recognition provides a user model of possible intentions and goals (but not beliefs), from the context of a plan library for a specific domain. However, some of the different responses, that Pollack attributes to a discrepancy of belief, can be combined and collapsed in providing an adequate explanation of violated constraints or failed preconditions. For example, the goal of preventing a person *X* from reading a file, may have a precondition of '*holds(¬ system_manager(X))*'. In explaining the failure of this precondition, the response should state that *X* is a system manager, but a better explanation might also include the reason for the precondition: that is, system managers cannot be excluded from file access.

2.1.4. Kautz's Plan Recognition

A theory of plan recognition that was more formal, general and powerful than previous approaches was proposed by Kautz [Kautz 87; Kautz and Allen 86]. Kautz's system allows the handling of concurrent actions and concurrent plans, abstract event descriptions and the sharing of steps between actions, and the ability to deal with disjunctive information. It can also handle both incremental and non-incremental recognition. An important motivation is that of making plan recognition more useful, so that important conclusions and predictions can still be made from a set of observations without uniquely identifying (or prematurely committing) to a particular plan. Since we adopt Kautz's plan representation in our examples, the following discussion highlights some of the basic aspects and terminology involved.

In Kautz's framework, plan recognition is viewed as deductive inference from a set of observations, an event hierarchy, and certain assumptions. The event hierarchy is a set of first-order predicate calculus statements about relationships between events, preconditions and effects of events, and equality and temporal constraints (an important part of Kautz's system is that of a temporal model to support reasoning about temporally complex situations). A *plan library* is represented by such an event hierarchy, but is more easily visualized and informative (for our purposes) in graphical form. Figure 2.1 is a graphical representation of a portion of Kautz's cooking hierarchy.

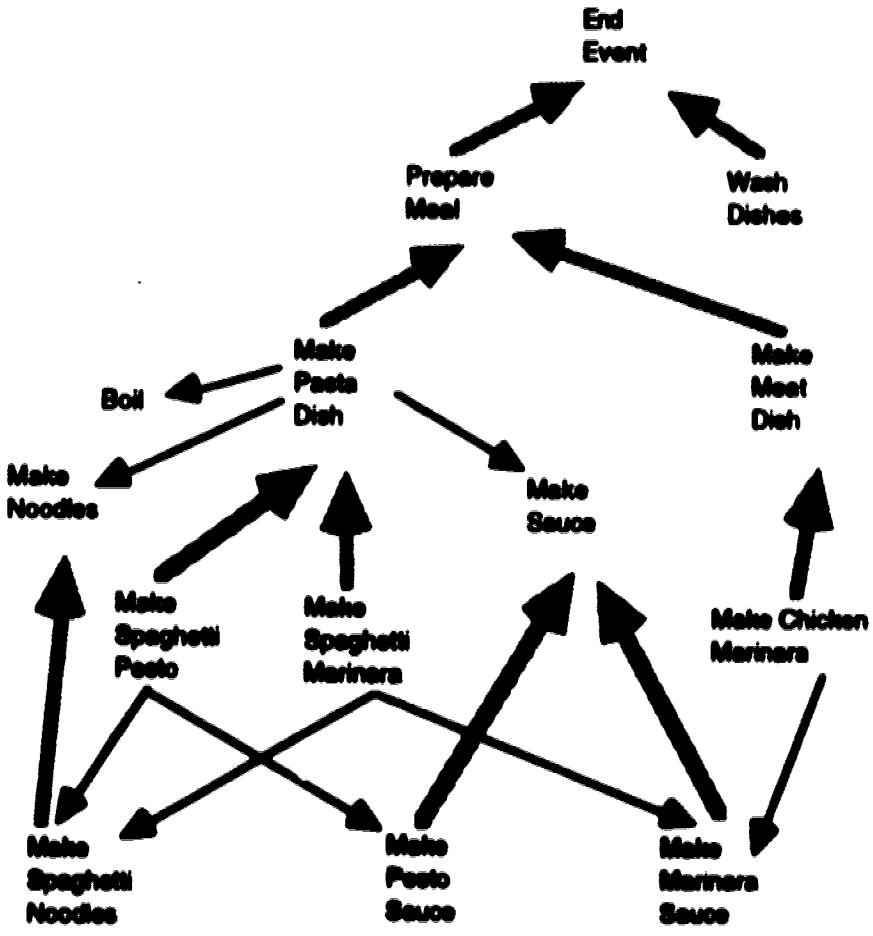

Figure 2.1 Portion of Kautz's Cooking Hierarchy

Figure 2.1 illustrates several important points. In this small plan library, there are two end events of *Prepare Meal* and *Wash Dishes*. These are events performed for their own sake and are not components of any other events. They represent the top-level actions in the domain being considered. The thick, gray arrows represent abstraction, or "isa", links. Thus, every *Make Pasta Dish* is also a *Prepare Meal* and every *Make Marinara Sauce* is also a *Make Sauce*. These links also reveal the specializations of an event, e.g., *Make Pasta Dish* and *Make Meat Dish* are specializations (or alternative ways) of preparing a meal. An event *may* have multiple abstraction parents. The thin, black arrows represent decomposition links. An event can be decomposed into events that are necessary actions, or subtasks, of the event in question. For example, the event of *Make Spaghetti Marinara* decomposes into the events of *Make Spaghetti Noodles* and *Make Marinara Sauce*. The event of *Make Pasta Dish* requires three substeps of *Make Noodles*, *Boil*, and *Make Sauce*. Decomposition links indicate the *uses* for an event, e.g., *Make Marinara Sauce* can be used to *Make Chicken Marinara* or can be used to *Make Spaghetti Marinara*. The hierarchy can be viewed as an "and/or" graph in which abstraction and decomposition links respectively correspond to "or" and "and" connections between nodes.

Kautz's plan recognition applies certain critical assumptions. An important concept is that the plan library is considered to be complete. This is based on two assumptions. The first is that the known ways of specializing an action are the only ways of specializing that action. The second assumption is that all ways of using an action as a substep of a more complex event are known, i.e., all possible reasons for performing an action are known. The event hierarchy is an exhaustive description of the relationships between events, and its completeness allows inferring a disjunction of possible plans that contain the observations. When more than one observation occurs, a heuristic can be used which assumes that as few end events occur as possible. This means that if different observations *can* be related (i.e., there are no constraint violations) to a single occurrence of an end event, then they are. Another important assumption is that of *disjointedness*. This assumption asserts that event types are disjoint, so that a particular instance of an event cannot be of more than one type. For example, an axiom of the hierarchy of figure 2.1 would be:

$$\forall x.\ \neg \text{MakePastaDish}(x) \vee \neg \text{MakeMeatDish}(x)$$

This does not mean that different event types cannot occur simultaneously, but that a particular instance of an event must be of one, and only one, type. A single instance of *Prepare Meal* means either a single instance of making a pasta dish or a meat dish, but not an instance of both.

There are certain problems that are not addressed by Kautz's formal theory of plan recognition. One is that the framework can only recognize plans derived from the initial plan library. Novel situations that require generalization or extension of the plan library cannot be recognized without the addition of some learning component. Another problem is that of distinguishing between likely and non-likely plans. Explanations of actions are treated equally within the framework. However, it can be argued that this is precisely what is desired in question-answering and advice-giving systems. Unlikely plans are plans that most people would not pursue due to some fault, or problem, with such plans. However, the users of these types of systems are expected to have a lack of knowledge in the domain being considered, and may just as easily formulate unlikely (and faulty) plans as well as likely ones. A plan recognition component, that only returned the most likely plans (or prematurely committed itself to such plans), would defeat an essential purpose of such advice-giving systems. However, if we wish to clarify and distinguish between

plans that have already been recognized, then the likelihood of such plans should be taken into account, providing that such likelihoods can be appropriately assessed and established.

Although Kautz does not deal with the issue of clarification, he does comment upon its importance, specifically with respect to medical diagnosis, but also in general terms:

> "Much of the research effort in medical expert systems has been on discovery procedures. The diagnostic system must take an active role, asking questions in order to narrow down the set of possible diseases. None of the work in plan recognition has dealt with this issue, yet it is critical if we are to build systems which can actively engage in a discourse or other task involving plan recognition."
> [Kautz 87, p. 17]

Our major interest in plan recognition is the result obtained, rather than the process itself. As an example of Kautz's plan recognition, we use the plan library given in figure 2.1, with a single observation of *Make Marinara Sauce*. The result is an event hierarchy that contains the observation and is a portion of the initial plan library. Figure 2.2(a) is a graphical representation of the result. The strongest conclusion that can be drawn is that the particular agent is either making spaghetti marinara or chicken marinara, but we do not know which without further evidence. A second observation of *Make Spaghetti Noodles* would allow us to conclude that the agent was making spaghetti marinara providing there were no constraint violations; for example, with respect to the agent and the time involved. This result is shown in figure 2.2(b).

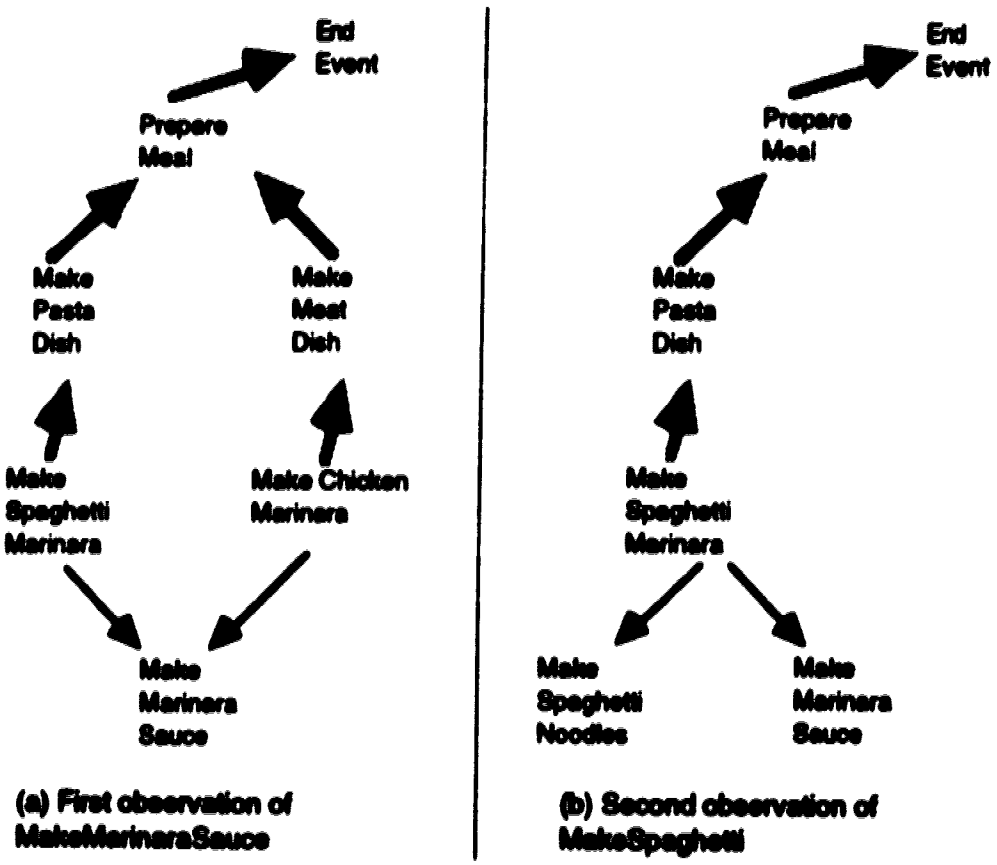

Figure 2.2 Recognized Plans From Observations

Throughout this background discussion, the concepts of plans, goals, and actions may appear blurry and confusing (e.g., Carberry has plans that consist of actions and actions that consist of plans, while Allen has plans with expectations and alternatives, and Pollack views plans as being a particular state of mind). Unfortunately, there is no formal definition of a *plan* that is widely accepted, and we do not attempt such an undertaking here. However, it is important for the topic and clear understanding of this thesis, that we establish some form of definition of what a plan is, so that there will be no confusion when we refer to their clarification. Chapter 3 provides such a definition for this purpose. However, since we adopt Kautz's representation of plans, figure 2.2 can be used to give a rough idea of our particular view of plans. In figure 2.2(a), the output of plan recognition consists of two possible plans, only one of which is being pursued by the particular agent. We view the end events as being the goals of plans and all other events as being specializations or component events to achieve a goal. The two plans of figure 2.2(a) both have the same goal of preparing a meal. We could describe the result by the following English sentences:

The agent intends to prepare a meal which is a pasta dish which is spaghetti marinara, by making marinara sauce.

(OR)

The agent intends to prepare a meal which is a meat dish which is chicken marinara, by making marinara sauce.

In figure 2.2(b), the result of plan recognition is a single plan:

> *The agent intends to prepare a meal which is a pasta dish which is spaghetti marinara, by making marinara sauce and by making spaghetti noodles.*

2.1.5. Better Cooperative Responses and User Models

Obtaining a user's plans and higher-level goals would allow a system to take them into consideration in providing a response to user questions. A plan recognition component that can determine the overall goals from a user's query would allow better responses (note that Kautz's framework does this; it basically classifies observations into plans to achieve certain top-level events or goals). It has been pointed out (e.g., [van Beek 87]) that cooperative systems, in order to be truly cooperative, should formulate a response that considers these overall goals, as well as the immediate goals of the user. This can be viewed as an extension of arguments previously discussed (by Joshi, Webber and Weischedel) regarding stated goals and intended goals, to include the higher-level steps leading to an overall goal. For example, consider the following responses to a user question of "Can I enroll in course X?":

- Yes (addresses the stated goal; the course is offered and there are no space limitations)
- Yes, but course X is not offered for credit (considers an immediate intended goal of obtaining credit for the course)
- Yes, but honors students should take course Y, instead of course X (considers a higher level goal involving a certain program of study)
- Yes, but your department recommends course Z over course X (considers another higher level goal)
- Yes, but you already have enough X-type credits, a better way would be to take a different course type (considers the overall goal of obtaining a degree).

These different responses provide warnings or alternative actions in light of the higher-level goals of the user. In order to provide such responses from a system utilizing plan recognition, the plan recognition component must be able to detect such higher-level goals. Also, the system must somehow be able to detect faults in the plans to achieve these goals, and be able to find better alternative actions (or plans) to achieve the same higher-level goals.

Obtaining the goals and plans of a user provides a certain knowledge of the user to the system. However, it should be noted that this is only a part of the information about the user that could be used by cooperative problem solving systems, especially those that attempt a natural language interface. Kass and Finin classify other information, such as the user's capabilities, attitudes, and knowledge or belief, in obtaining a user model [Kass and Finin 88]. Clearly, this extra knowledge would allow better responses. However, obtaining and utilizing such knowledge is outside the scope of this thesis.

Calistri-Yeh proposed a method for utilizing user models in handling plan ambiguity and plan-based misconceptions, in order to provide more robust plan recognition [Calistri-Yeh 91]. The key idea involves a probabilistic interpretation of the user model. Thus, a user model could explicitly represent the probability of the user having a particular plan, as well as the probability of plan-based misconceptions the user may have. Plan representation is similar to that of Kautz. Plans are stored in a plan hierarchy that forms a directed "and/or" graph in which "and" nodes form collections of

substeps and "or" nodes form collections of alternative (*alt*) steps. The arcs of the graph are each associated with a probability. Substep arcs all have a probability of 1.0, but alt arcs have probabilities that reflect the likelihood of their selection by the user. These values provide a means of preferring one plan over another. The point is that the particular user model can be isolated and altered by substituting a new set of probabilities. Calistri-Yeh also deals with invalid plans, not by explicitly storing them, but by assigning probabilities (and therefore, a user model) to certain features of a set of classified plan-based misconceptions that may cause a plan to be invalid. These misconceptions can be roughly divided into *constraint* misconceptions (e.g., violated bindings, preconditions, optimization, and temporal ordering) and *structural* misconceptions (which involve substituted, missing, or extra steps). Constraint misconceptions do not affect the nodes in the graph, whereas paths with structural misconceptions do not exist in the correct plan library and require altering the graph topology. Plan recognition is based on a best-first search from a single top-level goal and works by incrementally extending the partial solution paths (expanding the most likely path) until the user's query is reached. The result is a single plan that explains the user's query. Structural misconceptions are handled by proposing "reasonable" modifications to the graph structure. A controlling heuristic function, to determine the optimum path, is calculated from the probabilities of alternative steps from a parent node, and the probability of having a particular misconception with a particular plan.

There are some key differences between Calistri-Yeh's plan recognition and our own framework which is based on Kautz-style plan recognition. Currently, we cannot account for structural misconceptions by recognizing novel plans from a dynamic altering of the initial plan library. Calistri-Yeh's framework would require a large number of probability statistics; considering different probabilities for all alternative steps as well as those of misconception features (such as similarity, obscurity and complexity of a misconception in comparison to a correct plan). Establishing such numbers for each different user model would be a difficult task in any framework. Utilizing these probabilities, and heuristically resolving ambiguity by determining a single plan, always has the inherent problem of selecting the wrong plan (or worse, the wrong misconception). Given a set of plans with similar probabilities, there is a good chance that a wrong plan may be selected. In this case, the system must engage in some form of debugging process. This will involve aspects of the user realizing the mistake and informing the system, which in turn must recognize that it has provided an incorrect plan and must somehow backtrack and come up with the correct plan. Kautz's framework does not attempt to reduce ambiguity (nor does it over-commit to a single plan) unless there is evidence to support one plan over another through further observation of events. Calistri-Yeh's framework is meant to capture the fact that certain people are more likely than others to choose a particular plan. It relies upon somehow correctly classifying them into a particular user model of preferences or probabilities, before any plan recognition can be done, so that a particular plan can be heuristically determined. We take a somewhat different view, in that plan recognition does not assume any plan likelihoods and returns a set of possible plans in the domain. These can then be reduced by cases through further observations, and if necessary, by clarification, to provide an accurate user model of possible plans and goals, that can then be used to provide an appropriate response. The clarification process can use plan likelihoods, if such information is available, to determine the order of questions to ask the user; but not to over-commit the system to any one particular plan.

Before leaving this section on plan recognition and cooperative response generation, we should note that research in this area is still ongoing and quite active. There is still a wide gap between straight plan recognition theory and its practical implementation in applications. Kautz's framework may not be appropriate for every

domain. There may be applications in which it is not desirable to identify all possible plans that can explain an observation. Conversely, a complete plan library is too restrictive; an inference component to form complex and novel plans dynamically would be a tremendous asset, as would the capability of accounting for Calistri-Yeh's structural misconceptions. An excellent discussion of these shortcomings is provided by Goodman and Litman, who employ and modify Kautz's framework to provide a plan recognition component for an intelligent graphics interface [Goodman and Litman 92]. Similarly, for question-answering and cooperative problem solving systems, Kautz's plan recognition is not ideal. Besides the shortcoming of assuming a structurally correct and complete plan library, violated constraints and preconditions serve to prune the possible set of recognized plans in a strict Kautz-style framework, the purpose being to produce only the *correct* set of plans that will explain an observation. For advice-giving systems, we also want the incorrect plans, since these may indeed be correct and optimum in the mind of the user, and the violated constraints and preconditions can be explained in a response.

In light of the above discussion, we should emphasize that we do not advocate or support any particular method or style of plan recognition. The process itself does not directly concern us. We do assume that the result of plan recognition provides us with a possible set of plans, and these plans are in the form of a hierarchical structure that is similar to Kautz's representation (or an "and/or" graph, such as in Calistri-Yeh's representation). We also assume that an adequate response generator will provide an appropriate response related to a particular plan. Given these assumptions, the problem is what to do if the user of an advice giving system asks a question (expecting a response) and the recognized plans are ambiguous (i.e., there is more than one possible plan to explain the observations). One solution is for the system to clarify the plans with the user.

2.2. Clarification of Ambiguous Plans for Cooperative Response Generation

We view clarification as a process in which the system takes an active role in obtaining the user model (of plans and goals) by querying the user. This is a significant departure from systems which rely on passive observation of previous dialogue (or actions) and heuristically determine a single plan. Passive acquisition of the user's plan is preferable in order to minimize unnecessary dialogue. However, there are cases when the system should initiate a sub-dialogue (providing it has this capability) and directly acquire information from the user. There are two important considerations. The first is that of *when* to initiate a clarifying dialogue, and the second is that of *what* to ask the user.

2.2.1. Implications of Active Acquisition of User Models

Some of the implications involved in active user model acquisition were investigated by Wu [Wu 91] with respect to plan recognition in dialogue architectures. Wu points out that active acquisition (which we simply term 'clarification') can reduce overall dialogue length and increase the effectiveness of a system's responses. An important concept of Wu's system is that of an active acquisition goal (*AAG*). This is a goal that is held by the system to actively acquire knowledge from the user. The system is assumed to build a user model using passive acquisition, but occasionally adopts an AAG and hence, initiates an information-seeking subdialogue. Wu points out that there may be several reasons for the system to adopt an AAG, but focuses on those reasons that stem from a failure in the system's plan recognition or planning processes. Another important concept is that of the *utility* of different plans. The plans are assigned values for different utility attributes such as difficulty, productiveness, likelihood of success, over-specificness, and the degree of goal conflict. These values are used to compare different

plans and establish a partial order on them. A plan is deemed to be better than another plan if all of its attributes are superior.

Wu's system initiates active acquisition (or clarification) when the system adopts an AAG. These are adopted for several reasons. Exactly what is asked, depends upon the particular reason. Basically, the main reasons for generating an AAG are the following:

1. The plan recognizer produces no plan. In this case, an AAG is generated to identify an unknown user goal, or an unknown user misconception (and correct a misconception).

2. The plan recognizer produces multiple plans of non-comparable utility. In this case, an AAG is generated to disambiguate the user's intention or preference.

3. If no plan closely matches the specificness of known user goals, then an AAG is generated to determine a more specific user goal.

4. In comparing the difficulty of plans, if some plans have a precondition which cannot be determined, then an AAG is generated to determine the ease of achieving that precondition. An example of this would be that of:
   User: How do I get to the Marina?
   System: Do you drive?

One of Wu's example dialogues (in a domain of route planning) illustrates several features, and is provided as example 2.1 below:

(1) U: How do I get to the center of the bay?
(2) S: Why do you want to go there?
(3) U: I want to take a picture of the skyline.
(4) S: Is it sufficient to drive to Treasure Island, or is it necessary to take a cruise?
(5) U: No, a cruise isn't necessary.
(6) S: Then you should drive to the Bay Bridge and take the Treasure Island exit.

(7) U: What about Angel Island?
(8a) S: Why do you ask?
(9a) U: I also want to visit Angel Island.
(8b) S: There is no bridge to Angel Island, you must take a ferry.

Example 2.1

In the above example, using Wu's reasons for generating AAGs, the user's original query provides a goal which cannot be closely matched by any plan. In this case, reason number 3 results in an AAG to determine a more specific goal than that of getting to the centre of the bay. That is, a proper route (plan) cannot be determined without ascertaining a more specific goal. The reply discounts such plans as windsurfing to the centre of the bay, but leaves two possible plans with non-comparable utility. That is, a drive to Treasure Island may be just as good as a cruise, for the purpose of photographing the skyline. This leads to the generation of another AAG because of reason number 2. The reply of (5) results in the cooperative response of (6). The clarifying system questions of (8a) and (8b) are a result of AAGs for reason number 1.

Wu's system allows for a very broad scope, with respect to plan recognition as well as the active acquisition of the user's plan. For instance, the first reason for

generating an AAG implies that the plan recognition process is capable of handling novel plans. Also, it appears that the system is assumed to be capable of supplying values for the utility attributes of such new and novel plans. Although the dialogue of example 2.1 appears to be smooth and progressive with respect to providing a cooperative response, the situation could be different. Open-ended clarification questions such as 'Why do you want to go there?' or 'Why do you ask?' may result in replies that can increase the dialogue, because the system cannot account for them. For example, consider how the system should respond to a reply about an obscure goal the system does not know about. What does the system do when given a reply that it cannot account for? Should it repeat the question, say that it does not understand and ask again, or should it assume some default and carry on? The next clarification question, i.e., 'Is it sufficient to drive to Treasure Island, or is it necessary to take a cruise?', is also problematic. In this example the reply confirmed one of the two possible actions, but what if the reply had been a simple 'No' or a simple 'Yes'? In such a case, the exact intention of the user cannot be determined and a further question must be asked.

The model we adopt for this dissertation does not assume the recognition of new and novel plans, but can account for quite complex ones and can deal with the specificness of different plans, by representing them in an event hierarchy. The primary issue that we deal with is Wu's second reason for generating an AAG. That is, the case in which the plan recognizer produces multiple plans. We also restrict the system's clarification questions to certain formats, so that the user replies are restricted to 'Yes' or 'No' replies, or the selection of a menu item. There are several reasons for this. One is that such questions are clear and concise to the user. Neither the question nor the reply is ambiguous, and the user clearly understands the question. Such clarification questioning can be readily implemented in practical systems. We do not get involved with debugging issues of dealing with open-ended questions that ask the user to supply a goal or a sub-goal. Also, we can more easily compare different strategies to shorten the number of clarification questions, when such questions have a similar format.

2.2.2. Plan Critiquing and Clarification Dialogue for Cooperative Responses

In their work on plan ambiguity and cooperative response generation, van Beek and Cohen [van Beek and Cohen 91; van Beek *et. al.* 93] also propose solutions as to when the system should enter into a clarification dialogue and what the system should ask the user. They argue that a user's actions may often be consistent with more than one plan, especially after only one or a few utterances. Premature commitment to a single plan will mean that the system may commit to the wrong plan and require the ability to handle natural language debugging dialogues. Their hypothesis is that clarification dialogues are better than debugging dialogues. When the system commits to the wrong plan, users must recognize this from the responses, and realize that their plan differs. Further, users may not recognize that the system is pursuing the wrong plan, and they may be misled. Van Beek and Cohen argue that clarification dialogues need not be lengthy if the questions are carefully chosen. However, they also allow that there may be trade-offs between overcommitting in the plan recognition phase and engaging in lengthy clarification dialogues.

Given situations in which the result of plan recognition will usually be a disjunction of possible plans, van Beek and Cohen show how to deal with the ambiguity, in order to provide a cooperative response. In particular, they show that it is not always necessary to resolve the ambiguity to a unique plan. If the ambiguity does not matter, the system can continue answering the user's queries until further dialogue helps to clarify the exact plan the user is pursuing. If the ambiguity does matter, the system takes advantage

of the interactive and collaborative nature of dialogue, by asking the user clarification questions about pursuing certain actions that compose the possible plans.

An important part of the overall proposal is that of a *critiquing* component or procedure. Providing a cooperative response depends on the plan of the user. For example, a response should warn a user that a plan will fail because some precondition is not satisfied or inform the user if another plan has the same effect, but is better in some sense. The critiquing component analyzes the possible plans produced by plan recognition, and annotates them with their *faults*. That is, a plan containing an action that has a failure of a precondition is assigned a fault or label that indicates the failed precondition. The existence of a better plan would result in a fault that indicates this fact and provides another primitive action to suggest to the user. A plan having temporal inconsistencies in its actions would be assigned a fault annotation concerning the inconsistency. Also, a valid fault annotation could be that of *faultless* in the case that the plan has no legitimate faults. The fault annotation of a plan consists of the set of faults of the plan.

To determine when the system should actively clarify the plans with the user, the fault annotations of the plans are compared. For the purpose of comparing different plans, these fault annotations can be viewed to be similar to that of Wu's utility of plans in the previous section. If all possible plans are faultless, then their ambiguity does not matter to providing a cooperative response. That is, the response may simply provide a direct answer to the user's query. Similarly, if all plans have the same fault, or the same set of faults, their ambiguity does not matter. In this case, the response provides a direct answer to the user's question, but also adds additional information with respect to that fault annotation that is common to all of the possible plans. In essence, the ambiguity does not matter if all plans have the same fault annotation. Should all plans not have the same fault annotation, then their ambiguity matters to the response and the system enters into a clarification dialogue and asks the user clarifying questions.

Assuming that ambiguity matters, van Beek and Cohen propose a specific solution as to what to ask the user. Their proposal for clarification dialogue is tied to a hierarchical representation of plans and a hierarchical plan library. Plan recognition results in an event hierarchy, that is obtained from a plan library representing the domain. The recognized plans are annotated by the critiquing component. The clarification process receives as input a set of annotated plans and an event hierarchy that represents those plans. During clarification, the event hierarchy is traversed in a top-down manner. The key idea is that of asking the user about the highest level of possible events (or actions), and then checking to see if ambiguity needs to be further resolved. If it does, then the user is asked about the events at the next level down. This process repeats iteratively through the hierarchy of events until the ambiguity of the remaining possible plans no longer matters. That is, clarification dialogue terminates, and a cooperative response is provided, when all remaining plans have the same fault annotation. Thus, clarification is not used to determine the user's specific plan, but to resolve the ambiguity of possible plans to the point at which the ambiguity no longer matters for the purpose of providing a cooperative response. The individual clarification questions are of two types. Questions that require a 'Yes' or 'No' reply pertain to single events or actions. Menu-type questions require the user to select a choice from events that are related to a certain level of the hierarchy.

This thesis builds upon the work of van Beek and Cohen. We term their approach to be the basic approach. We enhance this basic approach and address some of its limitations. A detailed description and examples of the basic approach are provided in chapter 3.

Chapter 3. The Basic Approach and Its Limitations

Since the focus of this thesis builds upon previous work [van Beek and Cohen 91 ; van Beek *et. al.* 93], this section describes that work and its limitations, and establishes certain terminology. Before this description, we will first look at how the clarification process may fit into 'a larger picture', in some complete but arbitrary system.

3.1. Clarification as a Part of a Larger System

Our view of clarification is that it is a process which determines a user's plan from a set of possible plans. This is accomplished by actively asking the user about component parts of the plans until some form of plan-based response can be provided to the user. The systems or applications that may use this form of clarification of plans can vary widely with respect to purpose, architecture, organization and implementation. However, there are certain common denominators that such systems will have.

In such systems, plan recognition plays an important role. The role of plan recognition may be central such as in an information-seeking or advice-giving environment. On the other hand, the role of plan recognition may only be peripheral and used as an aid to accomplishing some other primary task. A good example of this is Genesereth's MACSYMA ADVISOR. In any case, the plan recognition component receives some form of input to be used as observations of actions that have taken place or are intended to occur. The form of this input is dependent upon the application and its purpose. For a question-answering system with a natural language setting, the input may consist of descriptions of actions obtained from textual key-board input, a menu-based natural language front end [Tennant 87] or even from a voice recognition component. For other applications the input may be a series of device-specific commands or some form of action-information based on visual or graphical images.

The plan recognition component may monitor its input of observations until some 'triggering mechanism' requires its output. The output of the plan recognition component will be a set of plans that are based upon the observations. The correlation between the observations and the output depends upon rules established by the application. Some of the concepts and problems that the specific application must handle are:

- The determination of context switching and how to deal with it. The problem here is one of consistency between several observations. A decision must be made as to whether the observations are related.

- If the context does switch, the application must decide whether to dismiss the recognized plans that are not immediately required or keep them in case some future observation fits in with these recognized plans. This is related to Carberry's ideas for the TRACK system.

- The recognition of invalid or faulty plans, as well as 'correct' plans. In certain applications, only correct and possible plans may be desired from plan recognition. In others, both incorrect ('buggy') plans as well as correct plans may be desired.

- A decision must be made about incoherent plans. That is, something must be done with respect to observations that cannot be accounted for by the plan recognition component, providing such observations are allowed to occur.

The output of plan recognition will be a set of possible plans that the user may be pursuing. If this set contains only one plan, the system can respond to the user. For an advice-giving setting, the response attempts to answer a user's question in a cooperative manner.

If plan recognition results in more than one plan, the system must handle this in some way. Heuristics may be used to reduce the number of plans to a single plan, if this was not already undertaken by the plan recognition component. Alternatively, clarification with the user may be undertaken if the system is interactive in nature. There is also the possibility of combining the two methods. The clarification process will be used, if necessary, to reduce the number of plans to the point at which a response can be provided. After a response is provided, the system resumes with its primary task.

For consistency, we assume, here, an advice-giving setting. The system is interactive and responds to user questions in a cooperative manner.

3.2. The Basic Clarification Model

The clarification process of the basic approach relies upon two critical factors. First, the recognized plans are represented by a hierarchy of events (actions). The second factor is that the plans have been analyzed (critiqued) prior to clarification. Our model therefore contains some form of critiquing process that annotates or assigns faults to the recognized plans. Some examples of these fault annotations are shown below:

1. Faultless.
2. Failure of preconditions: *preconditions*.
3. Temporally inconsistent.
4. There exists a better plan: *primitive actions*.

When the user asks a question, the system attempts to respond with an answer obtained from its knowledge of the particular domain. To be cooperative, the system not only provides a direct answer to the question, but may use plan recognition to determine a better and more informative response for the user (plan recognition may or may not be operating in the background prior to the user's question). Typical questions that are amenable to plan recognition, for the purpose of a cooperative response, are questions such as "Can I do *action X*?" or "Should I do *action X*?". Given such a question, the system should first determine if the action is possible. If the action is not possible, then the response should state this and provide an explanation. In this case, there may be no need for plan recognition and critiquing. If the action is possible, the response may also include information in light of the user's plan.

If the user's question is consistent with more than one plan, then there is an ambiguous situation. Our model handles this by first determining if the ambiguity matters for the purpose of providing a cooperative response. To do this, we compare the fault annotations of the recognized plans. For example, if all possible plans are faultless, we may respond to 'Can I ...' or 'Should I ...' questions with a simple affirmative response. If all plans are not faultless, but all have the same fault or faults, we can respond with a response that provides information about the fault(s). In essence, if the plans have the same fault annotation, then their ambiguity does not matter to providing a response. In

this case, the user continues interacting with the system even though the user's exact plan was not determined. Further interaction and observations may later reveal this exact plan, but its precise determination was not required to answer the user's question.

The ambiguity of the plans will matter if the recognized plans do not have the same fault annotation. In this situation, the clarification process begins. The system responds to the user's question with a question, in an attempt to disambiguate the plans. The clarification process continues up to the point at which ambiguity no longer matters (i.e., we do not attempt to determine a single plan). At the end of clarification, there may still be several ambiguous plans, but they will all have the same fault annotation and a response can be provided to the user's question. In practice, our model partitions the recognized plans according to their fault annotations. All plans of a particular partition will have the same fault annotation. To determine if ambiguity matters, we simply check the number of remaining partitions. Ambiguity will matter if there is more than one partition. The clarification process continues until only one partition remains.

During clarification, we ask the user about the possible plans. Based on the user's reply, certain plans will be eliminated from consideration. We keep asking about the remaining plans, and eliminating, until all remaining plans belong to only one partition. Determining what to ask the user is the focus of this thesis. When the plans are simple and few in number, their clarification does not present a significant problem. For example, in Allen's train station domain the user intends to either board a train to a certain destination or meet a train from a certain destination. These plans can be described by a single event or action such as 'Board-Train(X)' and 'Meet-Train(X)'. There is no higher-level goal in the domain, so in essence, the user's goal of boarding a train suffices as a complete description of the user's plan. To clarify, we simply ask the user about one event or the other. For instance, if the user asks a question about the time of the Windsor train, we might ask "Are you boarding the train to Windsor?". If the reply is 'Yes', the plan of meeting the train from Windsor is eliminated. If the reply is 'No', the plan of boarding the train to Windsor is eliminated. In this latter case, the only remaining possible plan is that of meeting the train from Windsor. A response can then be provided based upon the remaining plan.

When the possible plans are more detailed, more complex, and greater in number, their clarification becomes a significant problem. We want to respond with a cooperative response that answers the user's question, but we also want to do this quickly. That is, we want the clarification process to avoid long, tedious, and confusing dialogues with the user. In order to achieve this, the clarification process should follow certain guidelines:

1. The individual clarification questions should be concise and clear. The user should not be expected to spend an inordinate amount of time in reading the question, understanding the question, or pondering why the particular question was asked. The user's reply to any clarification question should be fairly automatic and without hesitation.

2. The number of clarification questions asked should be minimized, as far as possible. We should not ask the user any questions that do not serve to eliminate or prune the possible plans. We should attempt to ask the user those questions that will most resolve the ambiguity.

As an example of dealing with numerous complex plans, consider a course-advisor system in a domain associated with a particular university environment. Cooperative responses are provided to user questions of 'Can I enroll in course X?' or

'Should I enroll in course X?', or even the imperative command of 'Register me in course X'. The response to such questions must consider several factors such as,

- Does the course exist and is it currently being taught?
- Is there space available, or are all course sections full? If so, is there an alternative course that will fit the user's plan.
- Are there necessary prerequisite or co-requisite courses, and if so, does the user have them?
- Is the course an elective? If it is, are there any restrictions on electives that are imposed by the user's program of studies?
- How does the course fit in with the user's plan? Is the course suitable for the user's particular department of study? Is the course appropriate for the user's particular program, such as an honors or specialization program? Does the course fit the degree requirements of the user? Is the user even obtaining a degree?

Given that the system does not have prior knowledge about the user (for example, the user may be a high school student), plan recognition could produce a large number of possible plans for any one course, considering the different possible degrees, faculties, departments, and different programs of study. One possible plan may be that a user intends to take a certain computer science course to meet department requirements for an honors program in computing science, in order to obtain a BSc. degree. We could ask about such a plan by perhaps asking "Do you intend to obtain a BSc. degree with honors in Computing Science?". If the reply to this question were 'Yes' then clarification would be over. The response can take into consideration any faults of the plan. However, if the reply were to be 'No', we can eliminate that one plan, but must ask about another. We might ask about specialization in computer science or a major in computer science, before moving on to a different department. After asking about the different programs for each department in the faculty of science, we might switch to another faculty and again ask about the different departments and programs of that faculty. With a fair number of possible plans, asking about each one individually can lead to a large number of clarification questions before we obtain the user's plan.

The clarification process proposed by van Beek and Cohen is tied to a hierarchical representation of the plans that is similar to Kautz's plan representation. The hierarchy consists of the individual events that make up the plans and the links between these events. Clarification proceeds through this hierarchy in a top-down fashion from higher-level events toward lower-level events. Each clarification question pertains to a single event rather than an entire plan. The user is asked about the pursuit of a certain action, e.g., "Do you intend to enter Computing Science?". One advantage of this approach is that the individual clarification questions are short and clear. Another advantage is that entire groups of plans can be eliminated by a single clarification question. This clarification process continues until all remaining plans have the same fault annotation and a response can be provided.

The best way to describe the basic top-down approach is through the use of examples. Before this, we should first establish a common terminology and certain ground rules for the various aspects and concepts of the problem. In the following discussions, we basically consider two domains for our examples. The first is a cooking domain, which is small and is meant to be adjusted to illustrate concepts and ideas. The second domain is that of a course-advisor for first year university students. This domain provides a more practical setting.

3.3. Terminology

Throughout the rest of this dissertation, certain terms are used with enough frequency to warrant a clear understanding of their meaning with respect to the problem. This section is intended to provide a common ground for the intended meaning of these terms. For each of the following terms there may be a parenthesized list of equivalent terms intended to have the same meaning. We begin our terminology with the 'plan library'. Most other terminology stems from this.

- **Plan Library:** Plan recognition recognizes plans from a plan library. This library may be represented as a set of first-order predicate calculus statements. For our purposes, we deal with the plan library in a graphical representation. In this form, the nodes of the graph correspond to events or actions in the domain. The arcs represent the links between these actions. Our representation is similar to that of Kautz. Figure 3.1 shows the plan library for our cooking examples. Several of the following terms are in reference to the plan library shown in figure 3.1.

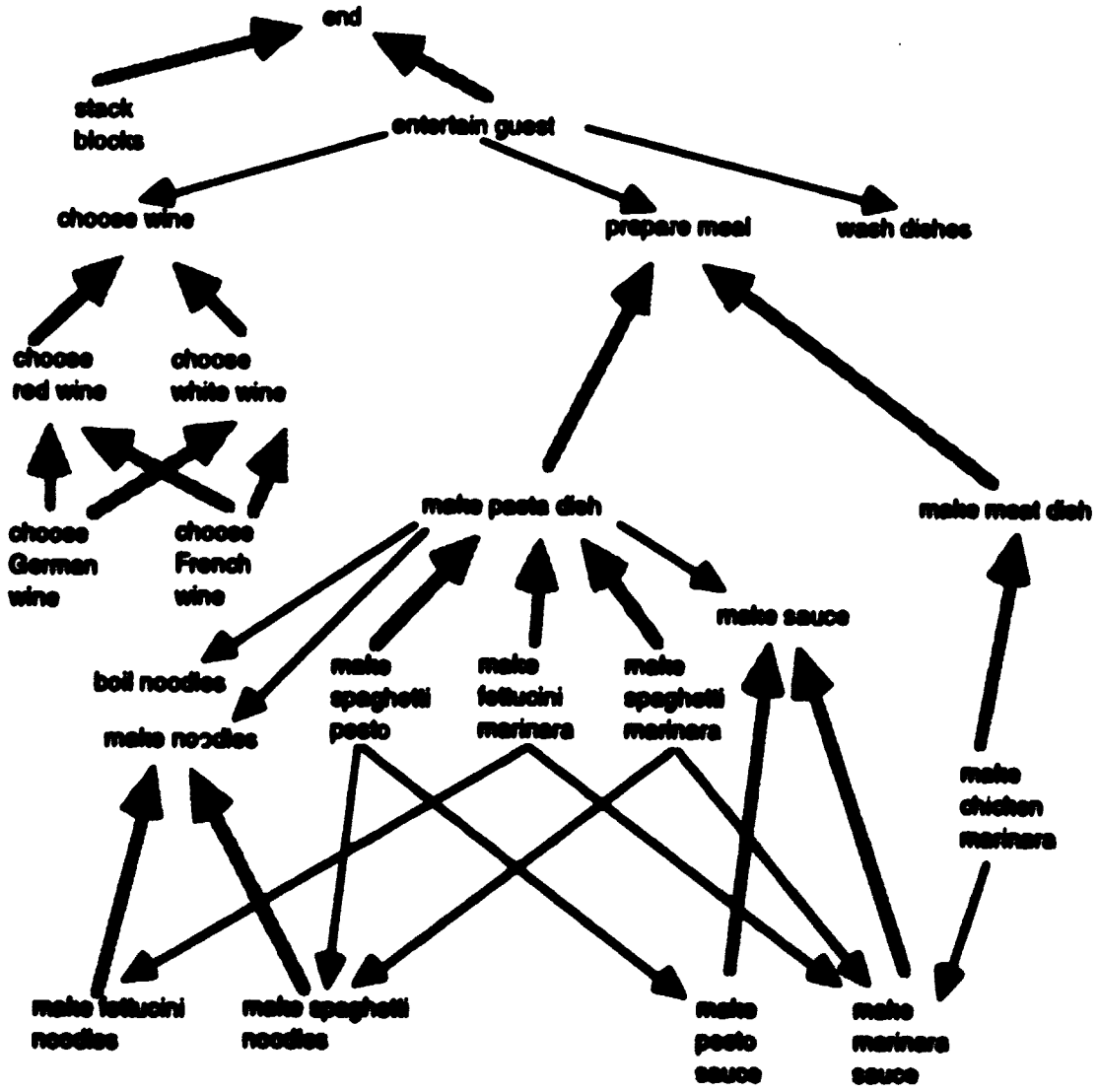

Figure 3.1 Plan library for cooking examples (Kautz 1987; modified)

Event (action): Events are the individual entities that form a plan. In the graphical form of the plan library, the nodes are the possible events within the domain. The events of the plan library are related by two types of links.

Abstraction Links: The thick upward pointing arrows represent abstraction (or "isa") links between events.

Decomposition Links: The thin downward pointing arrows represent decomposition (or "subtask") links.

Abstraction: An event that is an abstraction of other events, and has an abstraction link to those events. The abstraction is the event at the arrow-head of an abstraction link. *Make Pasta Dish* is an abstraction of *make spaghetti pesto*. Every *make spaghetti pesto* event is also a *make pasta dish* event. An event may have more than one abstraction.

Specialization (disjunct, disjunctive event, alternative): An event that is a specialization (or way of doing) some other event and is linked to that event by an abstraction link. The specialization is at the bottom of the abstraction link arrow. *Make spaghetti pesto* is a specialization (or way to) *make pasta dish*.

Decomposition Event (conjunct, conjunctive event, component): An event that is a necessary component or subtask required to perform some other event, and is linked to that event by a decomposition link. We mainly refer to these events as conjuncts. The conjunct is at the arrow-head of a decomposition link. *Make noodles* is a conjunct (or necessary step) of *make pasta dish*. Every event of *make pasta dish* requires an event of *make noodles*.

Uses: These are the events that make use of a conjunct and are linked to that conjunct by a decomposition link. The event is at the top (not the arrow-head) of a decomposition link. The event of *make noodles* can be used for the event of *make pasta dish*, so *make pasta dish* is a use of *make noodles*. Similarly, *make fettucini marinara*, *make spaghetti marinara* and *make chicken marinara* are all uses of the event of *make marinara sauce*.

End Event: This is a special event that serves to close the hierarchy. It is the root of the graphical representation of the plan library. The end event may only have specializations. All plans will include the end event. This event serves as the starting point for clarification.

Leaf Event: An event that has no specializations or conjuncts. We can view these events to be primitive actions in the domain. *Make spaghetti noodles* is a leaf event for the plan library of figure 3.1.

Goal: (top-level event): An event directly below the *end* event that is connected to the *end* event by an abstraction link. These events represent actions performed for their own sake. They are not specializations or components of other events in the domain. We can think of these events as being the ultimate goal of any particular plan. For example, with the plan library of figure 3.1, the goal of any plan involving an event of *make pasta dish* is that of entertaining a guest.

Event Hierarchy: We now reserve this term for the output of plan recognition. Since we generally adopt Kautz's theory of plan recognition, this will be a deductive process from a complete plan library in the domain. The output of plan recognition will be an event hierarchy in the form of a directed graph. This will essentially be a sub-graph of the graphical representation of the plan library. The events of the recognized plans will be represented by this sub-graph. It is important to distinguish this from the plan library itself. Therefore, we designate the term of event hierarchy to represent that hierarchy of events that is recognized (or output) from the plan recognition process. For example, with the plan library of figure 3.1, and an observation of *make marinara sauce*, our plan recognition component may give us the event hierarchy shown in figure 3.2.

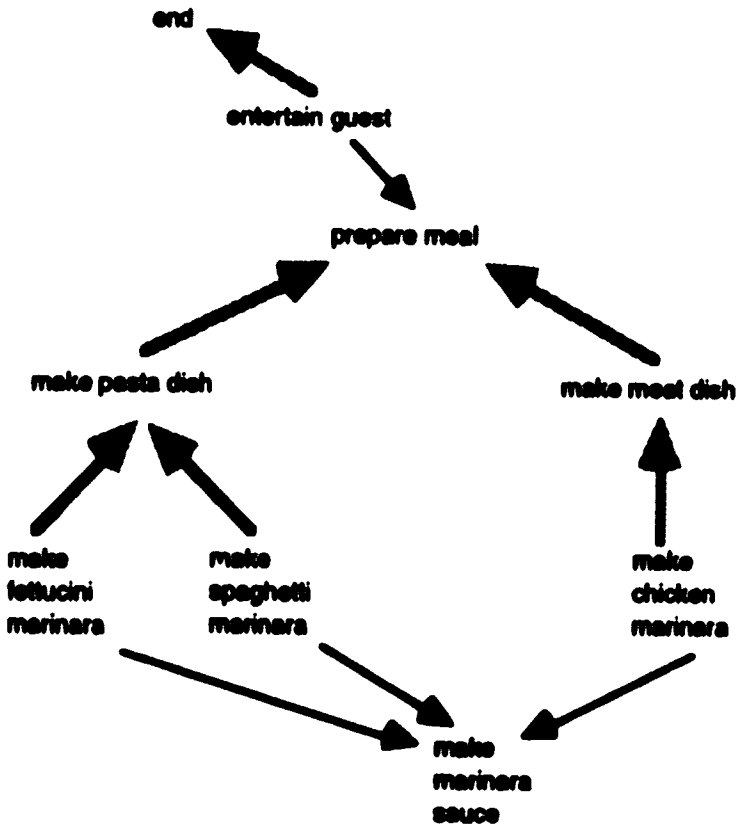

Figure 3.2 Event hierarchy from observation of marinara sauce

Plan: A plan is a collection of linked events. In our cooking domain, the action of making a meat dish is not considered to be a plan. It is simply an event. A description of a plan of the event hierarchy of figure 3.2 would be that of 'entertaining a guest by preparing a meal, which is a meat dish, which is chicken marinara, by making marinara sauce'. All plans will include the *end* event and some top-level event. The abstraction links in the event hierarchy distinguish different plans. The event hierarchy of figure 3.2 contains three plans as follows:

p1: (end, entertain guest, prepare meal, make pasta dish, make fettucini marinara, make marinara sauce).

p2: (end, entertain guest, prepare meal, make pasta dish, make spaghetti marinara, make marinara sauce).
p3: (end, entertain guest, prepare meal, make meat dish, make chicken marinara, make marinara sauce).

Disjunctive Branch Point: An event that has more than one specialization (disjunct). At such branch points, there are different possible alternative actions, which indicate different possible plans. The event hierarchy of figure 3.2 has two disjunctive branch points at *prepare meal* and *make pasta dish*.

Certain observations can be made with respect to the plan library and the event hierarchy (the recognized plans):

- Recognized plans may have different goals (i.e., different top-level events) or they may all have the same goal.

- Any event in the plan library may have any number of disjunctive events and conjunctive events linked directly below it. The only exception to this is the *end* event, which may only have disjunctive events.

- Any event in a single recognized plan may have no more than one disjunctive event linked directly below it. Thus, in preparing a meal x, x will either be a pasta dish or a meat dish, but not both.

- Any event in a single recognized plan may have more than one conjunct linked directly below it. These are the steps in the plan. Assuming another observation of making spaghetti noodles, the event hierarchy from plan recognition would contain a plan of '(end, entertain guest, prepare meal, make pasta dish, make spaghetti marinara, make spaghetti noodles, make marinara sauce)'. In this single plan, the events of *make spaghetti noodles* and *make marinara sauce* are both conjuncts of the event *make spaghetti marinara*.

3.4. Ground Rules

It is desirable to have clarification questions that are concise and clear. We restrict our questions to only asking about a single event or a menu of related events that are directly linked by abstraction to a single event. Furthermore, we only ask about disjunctive events in the event hierarchy of recognized plans. The conjunctive events do not serve to differentiate the possible plans. Asking the user about a conjunctive event is similar to asking the user if they intend to do something that is necessary to achieve an event that has already been established. For example, if we establish that the user is making a pasta dish, we do not ask if the user intends to boil the noodles. Every pasta dish will necessitate boiling noodles. Our concern is with which noodles are to be boiled. This does not mean that we should strictly avoid all conjunctive events in clarification dialogue. They can be used to provide more content to a question. The problem of minimizing clarification dialogue essentially boils down to selecting which disjunctive event to ask the user.

We restrict the user's reply to any clarification question. With respect to a clarification question about a single disjunctive event, the reply is expected to be a simple 'Yes' or 'No'. In a natural language setting, the reply to a clarification question may

provide other important information that is useful in further determining the user's plan. This is only beneficial if it occurs and the system is capable of handling it, but here we do not assume that this is the case. There is also the possibility of the user replying to a clarification question with another question. Again, we do not assume the capability of handling this situation. Our concern is with basic clarification strategies. Once the user asks a question of the system, we assume the system takes complete control until a response is provided. During this time, the user has no control, other than through a simple 'Yes' or 'No' reply to clarification questions. Should the clarification question be in the form of a menu of related events, the reply is expected to be a single selection from the menu.

We do not consider the response to be included in clarification dialogue. However, the length and clarity of any response is important to consider. The response should not include information that is irrelevant or useless with respect to the user's plan. In the basic top-down approach, a response is provided when ambiguity no longer matters as all remaining plans have the same fault annotation. The response addresses the faults of this annotation, which must include the faults of the user's particular plan. Our model, therefore, provides a response that is informative to the user, and does not supply other unrelated information.

3.5. Basic Top-Down Clarification

The basic top-down approach does not assume any observations from previous dialogue or information from any source other than a user's question. In general, at the point at which the user asks a question, plan recognition provides the possible plans that result from the question. The plans are then critiqued and given fault annotations. If ambiguity does not matter (i.e., all possible plans are faultless or all have the same fault(s)), a response is provided. Otherwise, we clarify by asking about certain disjunctive events, eliminating plans based upon the user's replies until ambiguity no longer matters, and a response can be provided. The basic top-down approach first asks about top-level events and then proceeds through the event hierarchy toward leaf events. The general algorithm for responding to a user question is as follows:

```
procedure GENERATE-RESPONSE(Query)
begin
        Check if Query is possible
        If Query fails then
                Output: "No, [Query] is not possible as [reason for failure]"
        else begin
                S ← PLAN-RECOGNITION(Query)
                S ← CRITIQUE(S)
                if AMBIGUITY-MATTERS(S) then
                        S ← CLARIFY(S)
                RESPOND(S) /* answer Query with a cooperative response */
        end
end
```

The clarification procedure (CLARIFY) maintains a current branch point (CB) and selects events to ask the user on the basis of this current branch point. Initially, the

current branch point is set to be the *end* event. Whenever the current branch point has only one child (or has only one remaining disjunctive event that has not been asked about), it is set to the next disjunctive branch point of the hierarchy in top-down fashion. That is, since ambiguity still matters, some other branch point lower in the hierarchy must still be clarified. All clarification questions pertain to single disjunctive events. A 'Yes' reply to such a question will eliminate all plans that do not contain the event. A 'No' reply will eliminate all plans that do contain the event. After the user's reply, some plans will be eliminated from further consideration as possible plans. After each clarification question we check to see if the ambiguity of the remaining plans matter. If it does, we ask another question that is based upon the current branch point; otherwise, we exit clarification with a set of plans that all have the same fault(s). The basic clarification algorithm is as follows:

```
procedure CLARIFY(S)
begin
        CB ← end   /* set the current branch point */
        while AMBIGUITY-MATTERS(S)
        begin
            if CB has only one child or one remaining event then
                CB ← next disjunctive branch point in top-down order

            Select a disjunctive event (E) of CB to ask the user about

            if user's answer is 'Yes' then
                    S ← plans having E
                    CB ← E
            else S ← S – (plans having E)
        end
        return(S)
end
```

3.5.1. Basic Clarification Example in the Cooking Domain

Assume that in our cooking domain, the user asks a question about marinara sauce. For example, the question might be "Can I make marinara sauce?" or "Is it okay to make marinara sauce?". From the plan library of figure 3.1, plan recognition gives us three possible plans (p1, p2, and p3) with the event hierarchy of figure 3.2. If the critiquing procedure found no fault with any of the three possible plans, they would all be annotated with 'faultless'. In this case, ambiguity would not matter and a response would be given, e.g., "Yes, you can make marinara sauce".

If the plans do not all have the same fault annotation, then we clarify with the user. Assume that the user's guest is a vegetarian and the system knows this. A precondition for the event of making a meat dish is that the guest not be a vegetarian. In this case, the critiquing of plan p3 (the plan involving a meat dish) will result in some fault annotation (f1) pertaining to a vegetarian guest. In this case the two pasta plans (p1 and p2) are faultless and plan p3 is not. Since ambiguity matters, we clarify. The current branch point is set to be the *end* event. Since this event has only one child (the specialization of *entertain guest*), the first task is that of finding the next disjunctive branch point. The next disjunctive branch point is the event of *prepare meal*. The current

branch point is set to be the event of *prepare meal*. Some alternative of the current branch point is selected to ask the user about. In this case, it does not matter which alternative (pasta dish or meat dish) is selected. Assume the user is asked about making a meat dish, e.g., "Are you making a meat dish?". If the reply is 'No', all plans involving a meat dish (p3) are removed from the set of possible plans. This will leave plans p1 and p2 as the remaining possible plans, in which case ambiguity no longer matters since both plans are faultless. If the user's reply were 'Yes', then the possible plans are those plans that do involve making a meat dish. This means that the only remaining plan is that of p3. Ambiguity will no longer matter, and a response is provided with respect to the fault annotation of f1. The actual dialogue might be similar to the following two cases:

Case 1:
User: Is it okay to make marinara sauce?
System: Are you making a meat dish?
User reply: No.
System response: Yes, you can make marinara sauce.

Case 2:
User: Is it okay to make marinara sauce?
System: Are you making a meat dish?
User reply: Yes.
(the remaining meat dish plan has a fault)
System response: Yes, you can make marinara sauce, but your guest is a vegetarian.

3.5.2. Basic Clarification Example in the Course-Advisor Domain

To provide a more practical and complex setting than the cooking domain, we assume a course-advisor system that is intended to give advice regarding first-year courses at a hypothetical university. It is assumed that the system does not have any background information with respect to the users of the system (the users are assumed to be new students who have not yet registered, but do intend to register in some degree program). The system can find faults with respect to a particular course and possible programs of study.

For our example, we assume that the user's question is about enrollment in a particular computer course (CMPUT 161) on word-processing applications for personal computers. The user's plan also involves obtaining a Bachelor of Science degree by specializing in computing science, but the system does not know this. A serious fault of this particular plan is that computing science students cannot obtain credit for the course CMPUT-161. Also, the department of computing science recommends certain computer courses for first-year students. We step through the dialogue of the example, providing explanations at each step, beginning with the user's question.

User: Can I take CMPUT 161?

The query is possible. From a large plan library for the domain, plan recognition gives us an event hierarchy that is partially illustrated by figure 3.3. In the figure, the events of particular interest that are closely related to the user's plan are shown. For clarity, other events in the hierarchy are not shown, but their existence is symbolized by the numbered diamond shapes. The number is meant to represent the number of possible plans which involve the event directly above the number. This provides a cleaner perspective about the number of recognized plans, without illustrating all their events.

For example, there are 12 possible plans that involve an Engineering degree, all of which require a computing science course. There are 11 possible plans that involve a Bachelor of Arts degree. In total, the event hierarchy of figure 3.3[1] represents 40 possible plans. Some of these plans may be faultless, but assume that most of them have differing fault annotations, to which the response provides a recommendation of an alternative computer course more suitable for a particular program. By enumerating and labeling the plans of the event hierarchy of figure 3.3, from left to right, the user's particular plan would be **p22**.

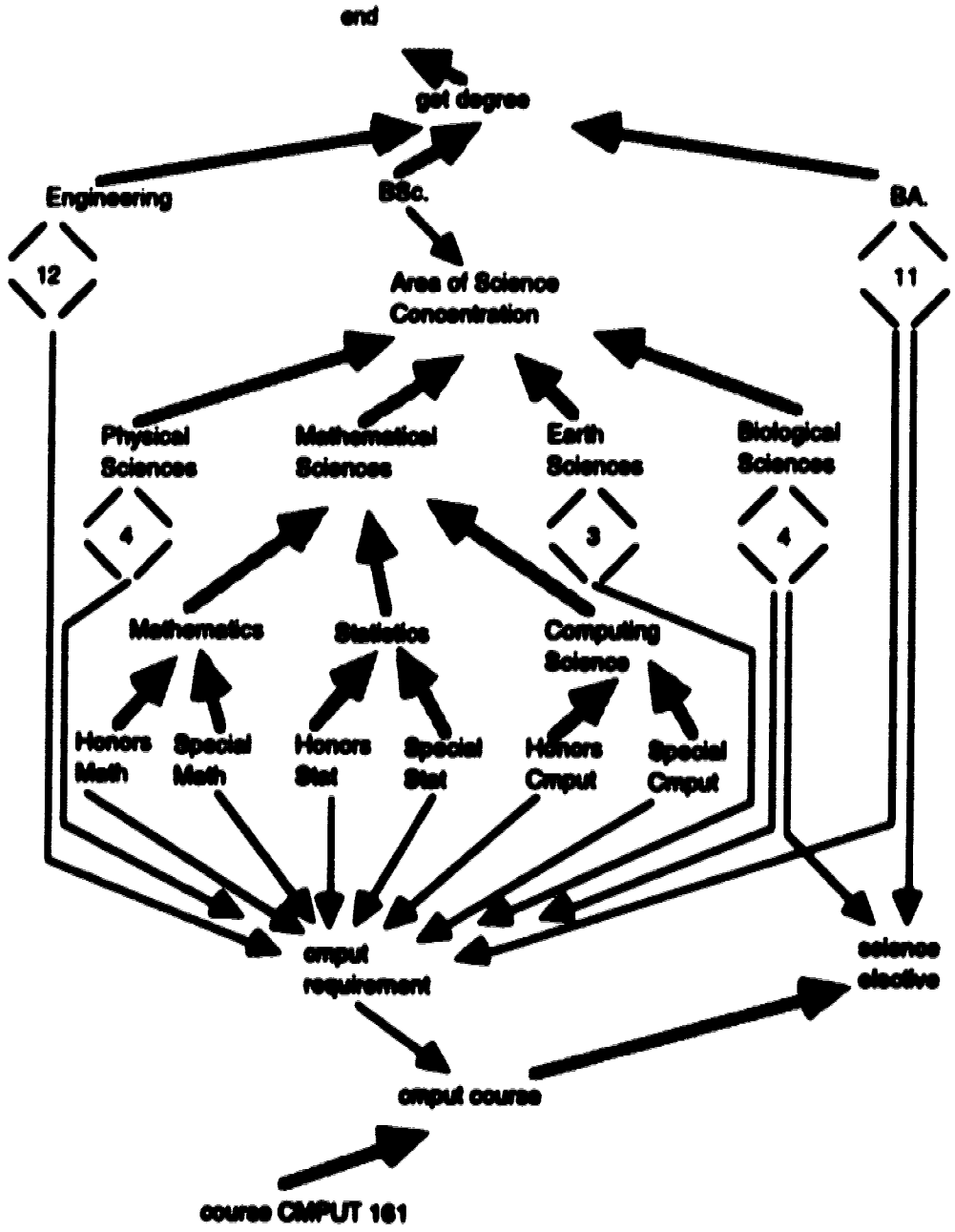

Figure 3.3 Event hierarchy for CMPUT 161 in the Course-Advisor

[1] As a side note, the specific alternatives of the area of science concentration are based upon actual groupings for general BSc. programs as given in the 1993/94 Calendar of the University of Alberta.

Given the event hierarchy of figure 3.3 the first disjunctive branch point is the event of *get degree*, which becomes the current branch point. This event has three disjuncts (or alternatives). For this example, we assume there are no heuristics to determine which alternative to ask first. By default, we ask about the alternatives in left to right order, with respect to their position in figure 3.3. This gives us our first clarification question.

System: Do you intend to obtain an Engineering degree?

User reply: No.

Based on the reply, 12 plans (p1 to p12) are eliminated from further consideration. The current branch point does not change and the next clarification question is with respect to the next alternative.

System: Do you intend to obtain a Bachelor of Science degree?

User reply: Yes.

Since the user's reply was a 'Yes', the remaining plans are now reduced to those plans involving a BSc. degree (p13 to p29). The remaining plans not involving a B.Sc. have been eliminated. These were the 11 Bachelor of Arts plans (p30 to p40). Also, since the reply was a 'Yes', the current branch point is set to the event of *BSc*, but since this event has only one child we determine the next disjunctive branch point (and new current branch point) to be the event of *Area of Science Concentration*. This event has four alternatives that effectively serve to partition the various departments of our hypothetical science faculty into four separate and distinct groups. Our next clarification question is about the area of physical sciences.

System: Will you be concentrating in the area of physical sciences?

User reply: No.

Based on the reply, 4 plans (p13 to p16) that involve the departments of physics and chemistry, are eliminated. The current branch point does not change.

System: Will you be concentrating in the area of mathematical sciences?

User reply: Yes.

The 'Yes' reply will reduce the possible plans to those having the event of *Mathematical Sciences* (p17 to p22). The current branch point is set to this event, which is also the next disjunctive branch point. Plans that involve the departments of geology and cartography (p23 to p25 of earth sciences) and the departments of genetics and microbiology (p26 to p29 of biological sciences) have been eliminated from further consideration. The unclarified events of the remaining six plans correspond to the portion of the event hierarchy shown in figure 3.4.

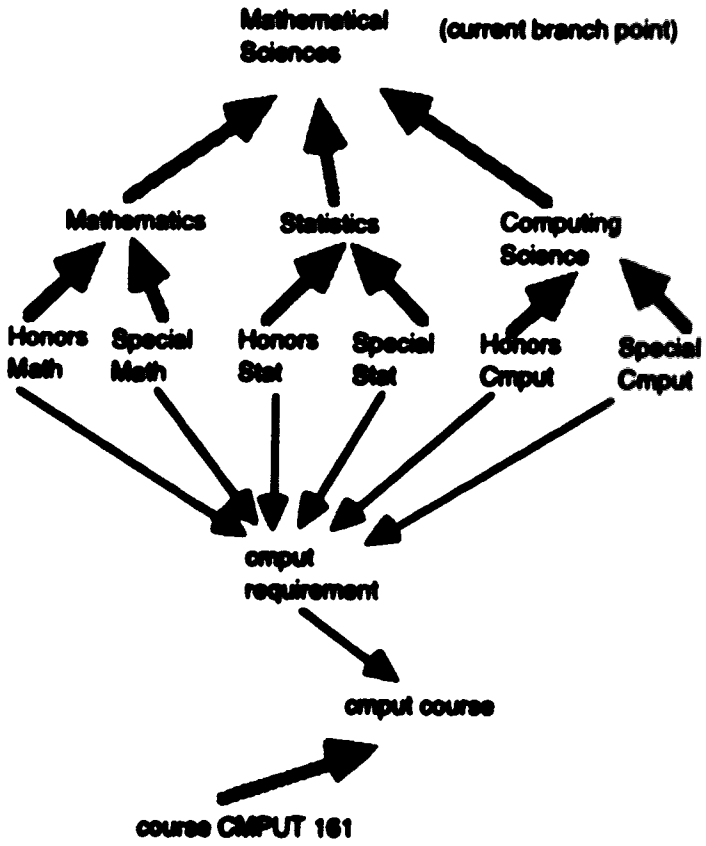

Figure 3.4 Mathematical Science portion of the event hierarchy of figure 3.3.

At this point in the clarification, ambiguity still matters. Assume that both our departments of Mathematics and Statistics recommend a different course (e.g., CMPUT 162) as a first-year computing science course requirement. This course is an introductory course to computer systems and involves some fundamental programming concepts. As such, it is preferable to the application specific course of CMPUT 161. The plans which involve either department have a fault annotation that addresses the unsuitability of the course of CMPUT 161. The two plans that involve the department of computing science represent different programs of study, i.e., an honors program and a specialization program (these are the only two types allowed by our hypothetical science faculty). Both programs have the same fault annotation. Therefore, of the six remaining plans, four have one type of fault annotation, while the other two have a different fault annotation.

In following the basic top-down approach, the next clarification question deals with a specific department in the area of mathematical sciences.

System: Do you intend to enter the department of Mathematics?

User reply: No.

The reply eliminates the two plans that involve the event of *Mathematics*. That is, the honors math program and the specialization math program. Ambiguity still matters for the remaining plans (p19 to p22). The current branch point does not change.

System: Do you intend to enter the department of Statistics?

User reply: No.

The two plans involving statistics (p19 and p20) are eliminated. The remaining possible plans (p21 and p22) both have the same fault annotation. Since ambiguity no longer matters, a response is given based upon the common fault annotation of the remaining plans.

> **System response:** Computing science students cannot obtain credit for the course CMPUT 161. The Computing Science department recommends structured programming courses such as CMPUT 163 and CMPUT 164 to meet first year computer course requirements.

The response addresses the faults of the two remaining plans. Note that if the two remaining plans had different fault annotations (e.g., CMPUT 163 is recommended for computing science honors students, while CMPUT 164 is preferred for specialization students), then the current branch point would change. Since the event of *Computing Science* is the only remaining disjunct of *Mathematical Sciences*, it becomes the next disjunctive branch point and the new current branch point. That is, the next clarification question does not ask about *Computing Science*, since all remaining plans must include *Computing Science* by default. Rather, the next clarification question would ask if the user were intending to enter the Honors Computing Science program.

The above example used a total of six clarification questions before providing a response to the user's original question. In light of the number of plans involved, we might consider this to be passable. The questions were short and clear, and we want to maintain this, but it would be desirable to find methods that can reduce the total number of clarification questions, if possible.

3.6. Improving the Basic Top-down Approach

As the starting point of this thesis work, the basic top-down clarification procedure was implemented in a trial Prolog program that we shall call the Response-Generator, and is described in [van Beek *et. al.* 93]. The plan recognition required was performed by a Lisp program written by H. Kautz (see [Kautz 87]). The output of plan recognition was a disjunction of possible plans that were then hand critiqued, annotated, and formatted. The possible plans (in the form of list structures of disjunctive events) then served as input to the Response-Generator. The clarification questions came from a database of queries for events in a small course-advisor domain about computer courses. The program provided several insights into the problem and some of these were dealt with in the implementation.

When the system does not directly ask about events of the user's intended plan, the clarification questions appear to jump to a different topic (from the user's viewpoint). That is, the user gives 'No' replies until the line of questioning suddenly switches context. For instance, in our example, 'No' replies to questions about an Engineering degree and a BSc. degree may directly lead to a question about a certain Arts program, rather than an Arts degree. In these cases of a *default* situation, the output of the Response-Generator contained a reassuring statement to let the user know exactly where the line of questioning was proceeding, e.g., "With respect to a Bachelor of Arts degree, ... ".

The implementation also experimented with a criterion for asking about the "most likely" group of plans. That is, with any current branch point having several alternatives, we want to first ask about that alternative most likely to be in the user's plan. This was accomplished by assigning *likelihood* numbers to the plans. In our course-advisor domain, these numbers may represent the previous year's enrollments. In keeping with our example, we might assume that there were no computing science students that took the course CMPUT 161 in the previous year. Thus, the likelihood number for the user's plan (p22) is 0. In considering a group of plans, the likelihood numbers of the individual plans are summed together to provide a likelihood for the group. Let the total for Engineering plans be 20, the total for all BSc. plans be 250, and the total of the eleven BA. plans be 650 (i.e., out of 920 students enrolled in CMPUT 161, in the previous year, 650 were in the Arts faculty). The criterion for selecting an alternative is to select that alternative with the highest likelihood total of all plans having that alternative. With these particular likelihoods, our first clarification question would be about a BA. degree, rather than an Engineering degree. Similarly, if the reply were 'No', then our second question would ask about a BSc. degree, rather than the Engineering degree.

The implementation of the Response-Generator also investigated the use of menus in querying the user. At each new current branch point, the alternatives of that branch point were enumerated and displayed. The user selected the disjunctive event being pursued. The remaining possible plans are always those plans that contain the event selected by the user. Note that this form of menu questioning maintains the individual clarification questions to be short and clear. All items of any single menu consist of single events, that are all direct specializations of the current branch point. Therefore, the individual menu items are closely related in context. We term this particular menu form to be a *homogenous* menu. This use of homogenous menus with our previous course advisor example would give us the following dialogue:

User: Can I take CMPUT 161?

System: Do you intend to obtain
1. an Engineering degree
2. a Bachelor of Science degree
3. a Bachelor of Arts degree

User reply: 2.

System: Will you be concentrating in the area of
1. Physical Sciences
2. Mathematical Sciences
3. Earth Sciences
4. Biological Sciences

User reply: 2.

System: Do you intend to enter the department of
1. Mathematics
2. Statistics
3. Computing Science

User reply: 3.
(At this point, ambiguity no longer matters)

System response: Computing science students cannot obtain credit for the course CMPUT 161. The Computing Science department recommends structured programming courses such as CMPUT 163 and CMPUT 164 to meet first year computer course requirements.

This menu approach offers several advantages over the 'Yes/No' type of questioning. The foremost advantage is that only a single query is required for each new disjunctive branch point. In our particular example, only three clarification questions were required instead of six. A less obvious advantage is that all possible events of any current branch point are explicitly presented. Users can more easily see where their particular plan "fits", when presented with all possible choices. Also, default situations do not occur, so there is no need to reassure users, who are always aware of their particular menu selections. One last advantage is that we do not require heuristics to determine the next alternative event to query the user about.

Considering the strong advantages of menu questions, it would seem reasonable to conclude that we should always use menus. However, there may be situations for which menu questions are not desirable or possible. For instance, the designers of a system with a natural language setting may not wish to compromise a "clean" natural language interface with the use of menus. Certain systems may simply not be able to use menus. A system using voice recognition and voice synthesis would find it difficult to use a menu style of questioning. A better example is that of automated telephone systems such as airline booking systems. Several of these systems currently exist, in which the system attempts to clarify a particular user's intention. The user's reply consists of pressing a certain number (or symbol) or a combination of numbers (and symbols) on the telephone. Although menu style questions appear to be asked by these systems, these questions are more in the form of sequential 'Yes or No' proposals. The following 'menu' is an example from an automated long distance calling system (to access the system, the user enters a zero followed by a seven digit telephone number):

This is *Company Name* long distance.

To charge this call to a calling card, enter the calling card number now!

For a collect call, press one, one.

To charge the call to another number, press one, two.

For person to person and other calls, press zero for the operator.

If the user intended to use a calling card and entered the calling card number after the word "now", the call is immediately put through. That is, the user does not hear the rest of the dialogue. In essence, the system interprets non-action on the part of the user (not pressing any buttons) as being a 'No' reply, and continues its dialogue. The exact ordering of each 'Yes or No' proposal (*question*) in the dialogue is of vital importance to the telephone company. Thus, even though this dialogue appears to present a menu to the user, it actually consists of a series of 'Yes/No' questions. As another example of automated telephone systems, the University of Alberta currently uses a telephone registration system. Users may add or delete courses concerning their curriculum. The system notifies the user if a certain course section is full, and it does suggest other course sections in this case, but this is basically the extent to which it provides a cooperative response. A more cooperative response might consider new course additions in light of a user's particular program (plan). In order to obtain the plan of a new user, the system might clarify with a series of 'Yes or No' type questions, to which the user could respond by pressing certain numbers on the telephone (similar to the long distance system). The ordering of such questions is important to reducing dialogue length. In these types of telephone systems, it is imperative to consider ordering aspects related to 'Yes/No' type questions. Since the 'Yes/No' type of clarification questions may serve an important purpose, the proposed strategies of the next chapter include certain methods for these

types of questions as well as menu questions. Also, when we do use menus, we do not restrict our questions to this format, but freely mix in 'Yes/No' types of questions for specific purposes in reducing the total number of questions.

The following chapter proposes certain concepts, methods, and algorithms, which are meant to co-exist with, and enhance, the basic top-down clarification method previously described. These methods address certain limitations of the basic approach. The following sub-sections briefly describe these limitations. The first sub-section deals with the general strategy itself.

3.6.1. Why not Bottom-Up?

It may be the case that a bottom-up approach would give us the user's plan much faster, but there are some complications involved in attempting to do clarification with a bottom-up approach.

The top-down method has one and only one starting point that is the *end* event. Also, the *end* event can only have specializations. It cannot have conjunctive children. In essence, we first establish the possible goal of the user, and then attempt to determine the user's exact plan (or a set of plans that contains the user's plan).

The first problem with using a bottom-up approach is finding which "bottom" we should start from. Given any arbitrary event hierarchy, we may have a large number of leaf events. The observation may or may not be a leaf event depending on the plan recognition used. In such a case, there is the problem of where to begin the clarification. Figure 3.5 shows a certain recognized event hierarchy from a single observation of *make sauce* . In this case there are only two leaf events.

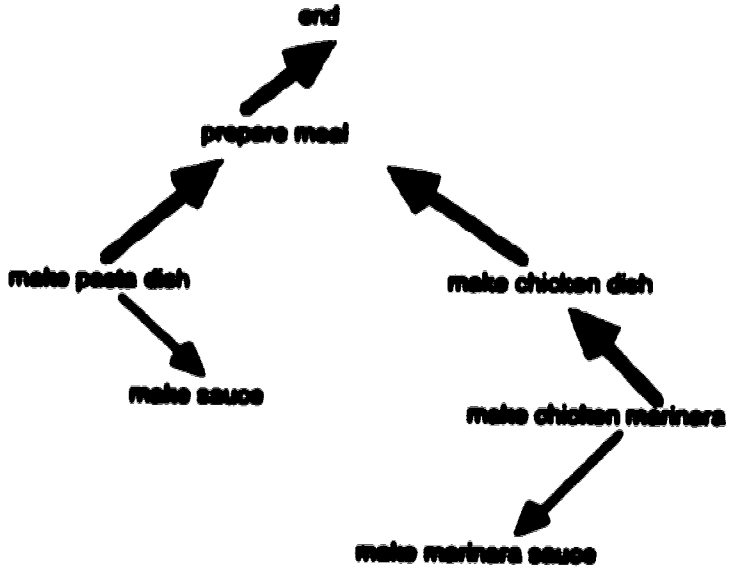

Figure 3.5 Event Hierarchy from "make sauce" observation

Figure 3.5 shows the recognized event hierarchy from the plan library of figure 3.1. In this plan library, the event of *make sauce* has two specializations of pesto sauce and marinara sauce. Only marinara sauce can be used for either a pasta dish or a chicken dish. Kautz's plan recognition program gives us these two plans; all different types of

pasta dish (such as spaghetti pesto or spaghetti marinara) are deemed to be redundant for the purpose of Kautz's plan recognition. However, modifying Kautz's program or using a different plan recognition method may allow the different specializations of a pasta dish to be output as possible plans from the same observation. In such a case, the challenge is determining which event to use as a starting point for clarification. Multiple observations further complicate this issue. There is also the problem of determining the next branch point with an upward progression in the hierarchy. In figure 3.5, the first (and only) branch point is the event of *prepare meal*. A 'No' reply to making a pasta dish would leave the meat dish plan. However, if there were several possible meat dishes (e.g., see figure 3.6), should the next branch point be the event of *make meat dish* (working top-down) or some 'bottom' of the meat dish branch?

Starting from the bottom may lead to other complications illustrated by the event hierarchy of figure 3.6.

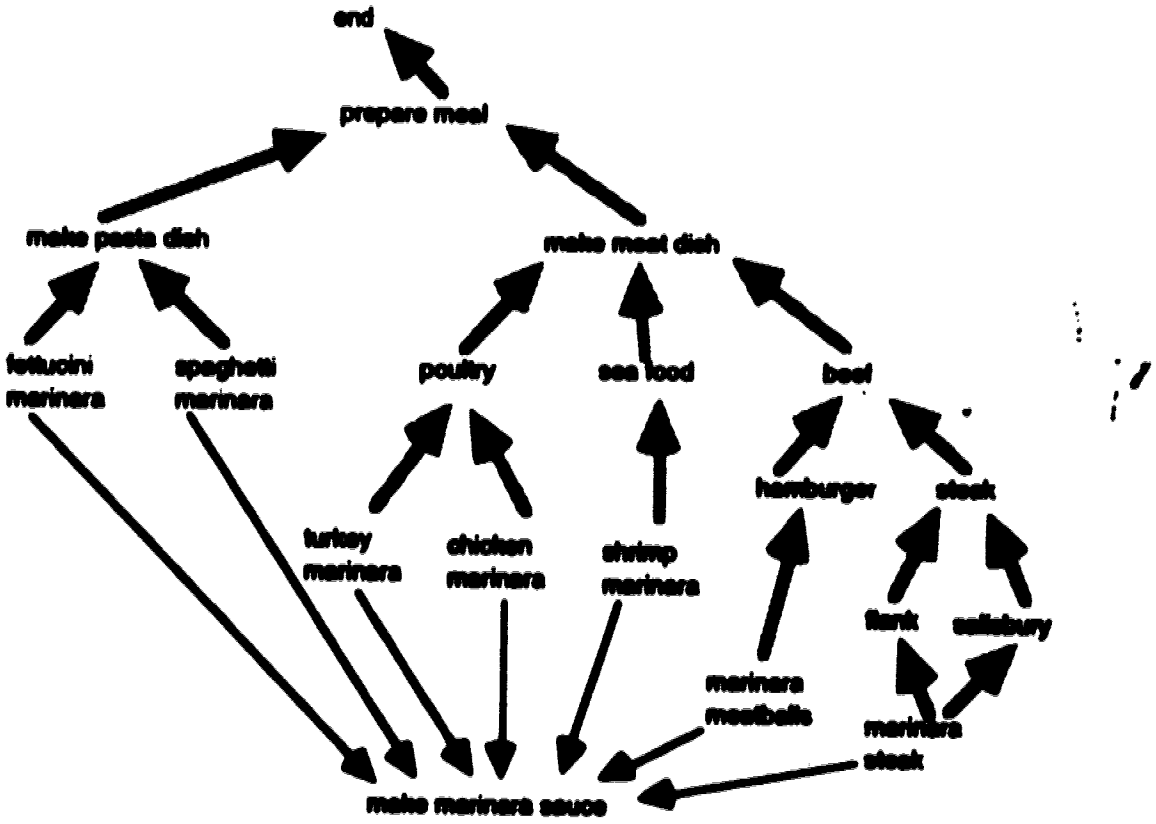

Figure 3.6 Hierarchy with a "bushy" bottom

Assuming we are given the recognized event hierarchy of figure 3.6, we do not have any problem with respect to a starting point for a bottom-up approach. There is only one leaf event of *make marinara sauce*. In this case, marinara sauce has seven possible different uses which are to be clarified. Obviously, a single menu question of these seven items will not be an homogenous menu (the menu items are not all direct specializations of a single event). If we are not using menus, then we enter into coherency and context problems of hopping back and forth between different groups of plans. For example, a possible dialogue might be:

1. Are you making spaghetti marinara? No.
2. Are you making chicken marinara? No.
3. Are you making marinara steak? No.
4. Are you making shrimp marinara? No.
5. Are you making turkey marinara? No.
6. Are you making fettucini marinara? Yes.

To avoid this, we must traverse up the hierarchy in order to group the various events and deal with the different groups one at a time. If we do this, we are in effect using a top-down approach. The above dialogue also shows that we can ask a lot of questions when the hierarchy has a 'bushy' bottom. If this were not the case, then we may be better off starting from the bottom, but we would somehow have to determine this first.

Proceeding from general questions in the domain toward more specific questions, as we do in the top-down approach, was an important design consideration for users of the MYCIN expert system [Buchanan and Shortliffe 84] dealing with medical diagnosis. An advantage of MYCIN's backward-chaining or goal-driven approach is that groups of questions asked by the system appear to be focused in evaluating particular diagnoses or goals.

In summary, there are certain advantages of the top-down approach over bottom-up:

1. It is simpler in a general sense.
2. We always know exactly where our next branch point will be with respect to the current branch point.
3. We can easily use homogenous menus at any branch point.
4. We maintain a 'focus' and consistency to our line of questioning.
5. We begin with high level goals and work toward specifics. It seems more reasonable to first establish *automobile* and then ask about *tires* or *headlights*, rather than establishing *tires* and then asking *automobile* or *tractor*.

The bottom-up approach may work better given a certain type of hierarchy, but we need to deal with the complications before it should be adopted as a general overall strategy.

3.6.2. Complex Plans

The basic top-down clarification method does not account for more complex plans. By *more complex*, we mean plans that contain events having more than one necessary component (or conjunct). These complex plans can result from single observations as well as multiple observations and introduce certain additional problems that must be dealt with. Multiple observations can occur from multiple queries as well as single queries. The systems that we consider may use dialogues for plan recognition, and not treat each user question as an isolated request. Thus, the ability to clarify complex plans that may often result from multiple observations is an important issue.

Our cooking domain provides the best example of problems with complex plans. Assume that we have a multiple observation with respect to making marinara sauce and a French wine. Both observations may occur in the user's question, e.g.,

User: Is French wine a good choice for a marinara meal?

With these two observations, assume that plan recognition gives us the event hierarchy of figure 3.7.

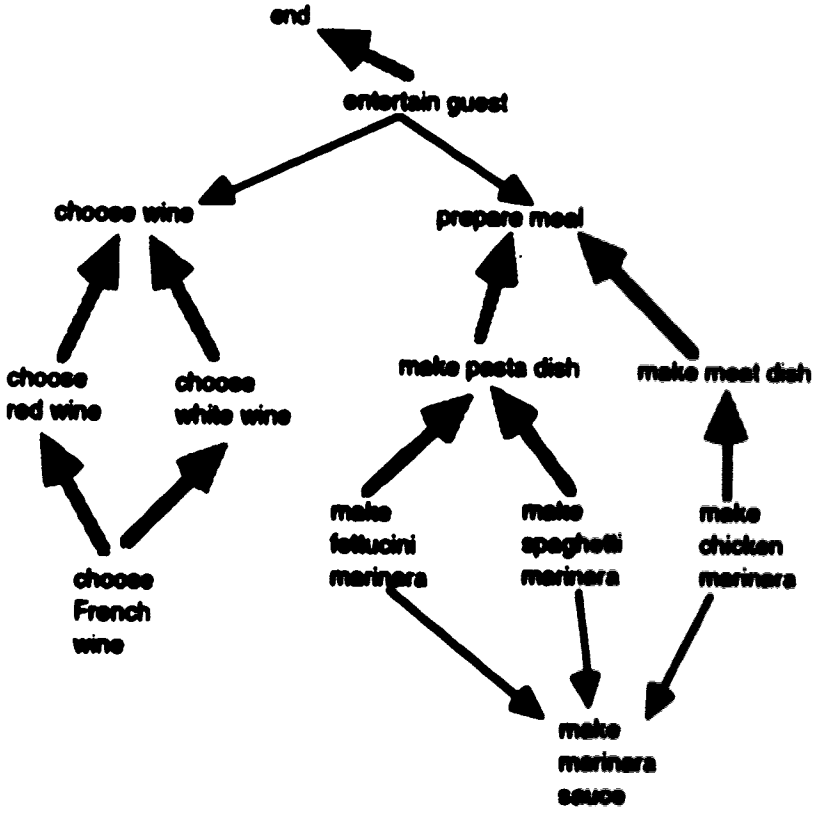

Figure 3.7 Event hierarchy with complex plans

The plans represented by the event hierarchy of figure 3.7 are all complex plans. We can view the hierarchy as being a compact representation of these plans. Each plan involves a French wine, which may be white or red, and each plan involves a marinara meal, which may be chicken marinara or one of the two pasta dishes. In order to achieve the goal of entertaining a guest, a wine must be chosen and a meal must be prepared. These are necessary sub-steps required to entertain a guest. However, there are two possible alternative choices for wine (red or white). Similarly, there are two alternative ways of preparing a meal, by making a meat dish or by making a pasta dish (and there are two possible ways to make a pasta dish). We can describe a certain single plan of this hierarchy as a plan to entertain a guest by serving a red wine which is a French wine, and by preparing a meal which is a pasta dish, which is spaghetti marinara. The plan of serving a red French wine with spaghetti marinara is completely separate and different from the plan of serving white French wine with spaghetti marinara. The event of *entertain guest* has two necessary components (or conjuncts), which are the events of *choose wine* and *prepare meal*. For clarity, should an event have more than one child, those child events that are conjuncts are termed to be sub-branches. Thus, the event of *choose wine* is a sub-branch of the event of *entertain guest*. We can view the action of choosing a wine, or preparing a meal, to be a sub-branch of a more complex plan intended to entertain a guest. The event of *make marinara sauce* is not a sub-branch of *make chicken marinara*, it is simply a conjunct. Every plan of this hierarchy has two sub-branches. Altogether, the event hierarchy of figure 3.7 represents six possible plans (each of the three possible marinara meals having two possible French wine choices).

If we apply the basic top-down approach to the event hierarchy of figure 3.7, the first problem we encounter is that of determining the first disjunctive branch point. That is, do we first ask the user about wine or about the type of meal. Assume we decide to go with wine.

System: Will you be choosing a red wine?
User reply: No.

This reply would leave us with only those three plans involving white wine. The next problem for this situation is again that of finding the next branch point. With the basic approach, we simply descended the hierarchy, but after clarifying which type of wine, there is no lower branch point in the given hierarchy. To further clarify the remaining plans we must pursue a new line of questioning for the other sub-branch of *prepare meal*. This necessitates keeping track of sub-branches of events. Since the paths descending from any sub-branch may contain other events with sub-branches (and these in turn may have paths containing events with sub-branches, and so on), our proposal for keeping track of them is to use a stack data structure called the *branchpoint stack*. In descending the hierarchy, we push sub-branches onto the branchpoint stack. If ambiguity matters, and we cannot determine a lower branch point in the hierarchy, we pop the branch point stack and continue looking for the next branch point from the result. In essence, we deal with sub-branches in a depth-first manner.

A significant problem with complex plans is that of unnecessary clarification of alternatives descending from a sub-branch. Given the event hierarchy of figure 3.7, let us again assume that the guest is a vegetarian. Let us also assume that our system can find no fault with serving either red or white French wine with any marinated meal. In this case, we have the following plans (in brief form) with their respective fault annotations:

label	*plan*	*fault annotation*
p1:	(red wine with fettucini marinara):	*faultless*
p2:	(red wine with spaghetti marinara):	*faultless*
p3:	(red wine with chicken marinara):	*vegetarian guest*
p4:	(white wine with fettucini marinara):	*faultless*
p5:	(white wine with spaghetti marinara):	*faultless*
p6:	(white wine with chicken marinara):	*vegetarian guest*

If we first ask about the choice of wine, we do eliminate half of the possible plans, but ambiguity will still matter. This will require a further clarification question with respect to the type of meal. The exact dialogue might be the following:

User: Is French wine a good choice for a marinara meal?
System: Will you be choosing a red wine?
User reply: No.
System: Are you making a meat dish?
User reply: Yes.
System response: French wine is a good choice, but your guest is a vegetarian.

Had we first asked about the type of meal, ambiguity would no longer matter and a response could be given. In this case, the dialogue would be:

User: Is French wine a good choice for a marinara meal?
System: Are you making a meat dish?
User reply: Yes.
System response: French wine is a good choice, but your guest is a vegetarian.

In this latter case, the dialogue is reduced by one clarification question. The problem here is not which sub-branch to select first, but whether or not to pursue any particular sub-branch. In this situation, the choice of wine does not affect the response (it does affect the number of possible plans, but not the response). The faultless plans include plans having white wine as well as plans having red wine. The type of wine does not matter for a response to a faultless annotation. Similarly, of the two plans having chicken marinara, one has red wine and the other has white wine, but both plans have exactly the same fault, which will result in the same response to that fault. Again, the choice of wine does not affect the response. In general, if we can determine that alternative events descending from a sub-branch are not required to resolve the ambiguity to the point of providing a response, then we can skip the branch points below that sub-branch (and hence, the clarification questions), and continue with the next sub-branch. In essence, we want to determine if we can resolve the ambiguity without pursuing further events below the sub-branch. A method for doing this is proposed and outlined in the following chapter.

3.6.3. Skipping Branch Points in Diamond Configurations

In certain cases, we may have situations in which paths of the event hierarchy diverge from a particular *disjunctive* branch point and later converge on some event lower in the hierarchy. We term these configurations to be *diamonds*. The *diamond top* is the particular disjunctive branch point event, while the *diamond bottom* is that event at which the plans first converge. The paths that connect the diamond top to the diamond bottom are the *intervening paths* of the diamond. Figure 3.8 illustrates a portion of an event hierarchy with a simple diamond configuration.

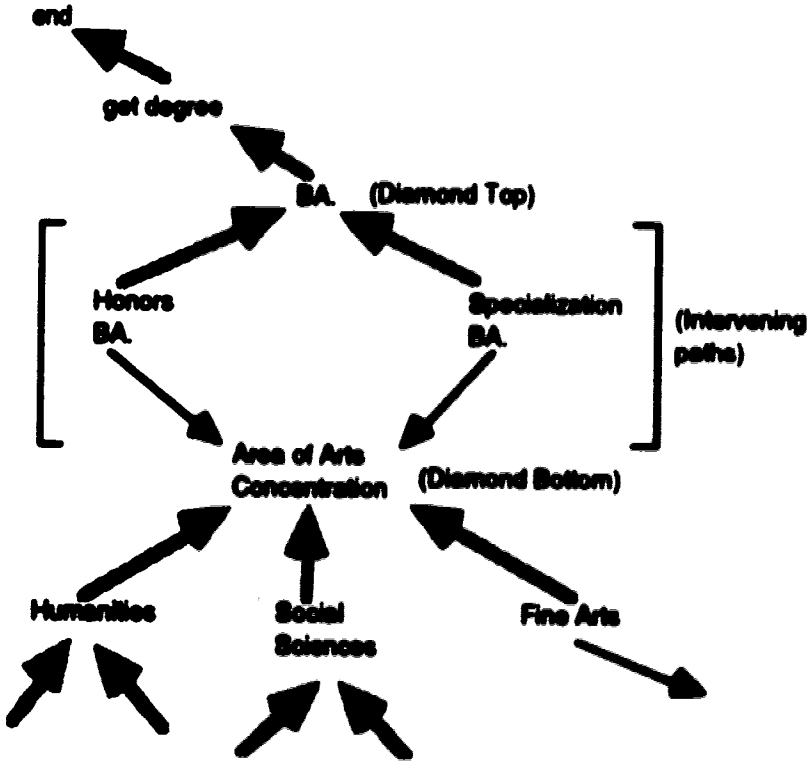

Figure 3.8 A Diamond Configuration

These diamond configurations allow us the opportunity of skipping branch points from the diamond top to the diamond bottom. As an example, assume that the current branch point is the event of *BA* degree in figure 3.8. Also assume that all remaining plans have fault annotations peculiar to the area of Arts concentration. So all plans that involve Fine Arts have the fault annotation of f1, plans involving the Humanities have a different fault annotation of f2, and all plans involving the Social Sciences have a fault annotation of f3. The basic top-down approach would first clarify the alternatives of the *BA* event, and then proceed from there to the branch point of *Area of Arts Concentration*. The ensuing dialogue may be:

System: Will you be entering the Honors BA. program?
User Reply: Yes.
System: Will you be concentrating in the Humanities?
User Reply: Yes.
System Response: *(Addresses the fault annotation of f2).*

By skipping branch points to the diamond bottom at which the plans converge, the dialogue would be:

System: Will you be concentrating in the Humanities?
User Reply: Yes.
System Response: *(Addresses the fault annotation of f2).*

In this case, the particular program (Honors or Specialization) does not matter to the response and did not require clarification. All Honors plans involving the Humanities have the same fault annotation as do all Specialization plans involving the Humanities. The same is true for the other areas of concentration. The key problem in skipping branch points is that of determining if the ambiguity can be resolved to the point at which it no longer matters, without resorting to the clarification of intervening events. That is, if a response can be provided by only clarifying events that occur "below" or "after" the diamond bottom, then we are able to skip branch points from the diamond top to the diamond bottom. Although the particular configuration of figure 3.8 is simple with few and short intervening paths, this need not be the case. Special problems, relating to branch point skipping, concern aspects of when to look for the possibility of skipping and finding the diamond bottom. That is, we do not want to search for a diamond bottom and the possibility of skipping at every new current branch point. The strategies and methods proposed for skipping branch points in diamond configurations are almost the same as that for determining if events descending from a sub-branch are required for a response, and are outlined in the following chapter.

3.6.4. Ordering the Disjunctive Events of a Branch Point

If we do not use menus and do not have access to statistics for the likelihood of certain plans over others, then we have the problem of ordering the 'Yes/No' clarification questions about the disjunctive events of any current branch point. The basic approach does not address this problem. In the absence of likelihood, the exact order of these questions may still have a significant affect on the total number of clarification questions required. We propose a set of rules to determine the ordering of questions about disjunctive events of the current branch point. These rules are based upon the *fault partitioning* resulting from the different annotations of the remaining plans, and are outlined in the following chapter.

3.6.5. Exploiting the Fault Partitions of Recognized Plans

In our general clarification process, the possible plans are partitioned, according to their fault annotations, before clarification questions begin. For the remainder of this thesis, it will be useful to think in terms of these fault partitions of the recognized plans. All plans having the same annotation will belong to one specific partition. Clarification questions reduce the number of possible plans. When a fault partition no longer contains any plans, that partition is eliminated. Ambiguity no longer matters when there is only one remaining partition (all remaining plans belong to that partition and have the same fault annotation). If at any time, there is more than one partition, then ambiguity matters. With this viewpoint, the faster we reduce the number of fault partitions, the sooner ambiguity will no longer matter. That is, clarification questions that eliminate partitions, as well as plans, are those questions we want to ask. One problem with asking such questions, is that we need to find the critical key events that will eliminate partitions. We do not restrict these events to be the direct alternatives of the current branch point. Also, we ask about these events with a 'Yes/No' type of question, whether we are using menus or not.

As an example, we can again use the course advisor domain and the event hierarchy of figure 3.3. The only change is with respect to the computer course. We assume all computer courses result in the same recognized plans, the only difference being the exact type of computer course, which will affect the fault annotations of the plans. For this example, assume that the user asks about the computer course CMPUT 165, a course on formal systems and logic that has both a prerequisite and a co-requisite computer science course requirement. Computing science students will meet these requirements, but other students may not. Let us assume that the 40 possible plans are partitioned into two fault partitions. One partition (partition-1) contains two faultless plans, i.e., the honors and specialization computing science programs. The other partition (partition-2) contains the 38 other plans, each of which has the same fault annotation with respect to the prerequisite and co-requisite requirements of CMPUT 165. Assume that the user's plan involves entering an honors program in statistics. The basic top-down approach (with menus) would give us the following dialogue:

User: Should I enroll in CMPUT 165?
(The query results in 40 plans divided into 2 partitions.)

System: Do you intend to obtain
1. an Engineering degree
2. a Bachelor of Science degree
3. a Bachelor of Arts degree

User reply: 2.
(The reply eliminates Engineering and Arts plans, leaving 17 possible Bachelor of Science plans which are still divided into two partitions.)

System: Will you be concentrating in the area of
1. Physical Sciences
2. Mathematical Sciences
3. Earth Sciences
4. Biological Sciences

User reply: 2.
(The reply results in 6 remaining plans which all involve Mathematical Sciences, but we still have two fault partitions. Partition-1 still has two plans, but partition-2 now contains only 4 plans)

System: Do you intend to enter the department of
1. Mathematics
2. Statistics
3. Computing Science

User reply: 2.

(The reply eliminates all plans involving mathematics and computing science, leaving only two statistics plans. Partition-1 is now empty and can be eliminated, leaving only partition-2, with the two possible plans. Since there is only one remaining partition, ambiguity no longer matters.)

System response: The course, CMPUT 165, has a computing science prerequisite requirement and a computing science co-requisite requirement.

Although the basic top-down approach can eliminate a large proportion of the possible plans with each clarification question, there is the possibility of doing better, by considering the remaining fault partitions. Instead of asking about an alternative of the current branch point, we may consider asking about some other disjunctive event that is shared by all the plans of a particular fault partition. In this case, partition-1 contains two plans that share the event of *computing science*. By asking about this event first, consider the following clarification dialogue:

User: Should I enroll in CMPUT 165?

(The query results in 40 plans divided into 2 partitions.)

System: Do you intend to enter the department of Computing Science?

User reply: No.

(The reply eliminates all plans involving computing science. This eliminates partition-1, leaving the other partition with 38 possible plans. But ambiguity no longer matters.)

System response: The course, CMPUT 165, has a computing science prerequisite requirement and a computing science co-requisite requirement.

What if the user's reply were to be 'Yes', i.e., the user did intend to enter the department of computing science, rather than statistics:

User: Should I enroll in CMPUT 165?

(The query results in 40 plans divided into 2 partitions.)

System: Do you intend to enter the department of Computing Science?

User reply: Yes.

(The reply eliminates all plans which do not involve computing science. This eliminates partition-2, leaving partition-1 with the remaining 2 possible plans. Ambiguity no longer matters.)

System response: The course, CMPUT 165, is a good choice.

With this particular example, a single clarification question is sufficient for providing a response, to either a 'Yes' or 'No' reply. It is important to note that, of all possible events in the given event hierarchy, only the event of *computing science* could eliminate a partition, without considering the user's exact plan. For instance, had the user considered an Arts degree, then first asking about an Arts degree would also have required only one clarification question. But asking about the event of *computing science* will eliminate at least one partition no matter what the user's plan actually is, and since

there are only two partitions, ambiguity will no longer matter after any reply. The event is held by all of the plans of partition-1, but is not held by any of the plans in partition-2, so it serves to isolate a single partition. In this case, we view the event of *computing science* to be a ***key event***. In the following chapter, we discuss the use of such key events, as well as some of the problems and limitations associated with them. We also extend the concept to that of isolating more than one fault partition.

3.6.6. Pointless Clarification Questions

With the basic approach, the clarification process terminates when ambiguity no longer matters. In certain situations, it may be desirable to stop further clarification, providing a response, before this termination condition. The reason for this is that further clarification is somewhat pointless to the user, even though ambiguity still matters to providing a proper response in light of the user's exact plan. If we ascertain that the user is intending to pursue a certain event that should not be pursued, then clarifying possible alternatives of that event, to provide a more precise cooperative response, can be viewed as being pointless and even uncooperative. We should stop further clarification and directly warn the user about the event.

As an example, consider our cooking domain with an event hierarchy that includes several meat dishes (see figure 3.6). The user is making marinara sauce for a meal to entertain a guest. Again, the guest is a vegetarian, but the user does not know this. Assume the different types of meat dishes have several different types of faults so their ambiguity matters to providing a response. In this situation, we might have the following dialogue:

User: Is it okay to make marinara sauce?

System: Are you making a meat dish?
User reply: Yes.
System: Do you intend to make a poultry dish?
User reply: No.
System: Do you intend to make a sea food dish?
User reply: No.
System: With respect to a beef dish, will you be serving steak?
User reply: Yes.
System: Do you intend to choose flank steak?
User reply: Yes.
(Assume that ambiguity no longer matters at this point)

System response: Yes, you can make marinara sauce, but your guest is a vegetarian, and you are out of flank steak, but the corner store has some.

We can imagine the user being upset with this dialogue, after reading the response. The user may be wondering why the system kept asking about different meat dishes, when a meat dish should not be made in the first place. The additional information about flank steak might be appreciated, but is pointless in light of the user's goal of entertaining a guest. A better dialogue would have been:

User: Is it okay to make marinara sauce?

System: Are you making a meat dish?
User reply: Yes.
(Ambiguity still matters, but the meat dish event should not be pursued)

System response: Yes, you can make marinara sauce, but your guest is a vegetarian.

With the above dialogue, ambiguity still matters. That is, the remaining plans have different fault annotations. Poultry dishes may have certain faults different from beef dishes. Plans involving hamburger may have certain faults that are different from plans involving steak. However, all remaining plans do include the event of *make meat dish*, an event that should not be pursued by the user. The event should not be undertaken because it has a rather serious fault associated with it. Because of a violated precondition, all plans containing the *make meat dish* event have a fault annotation that includes a fault concerning a vegetarian guest. This particular fault is of such a nature that the event associated with it should not be undertaken.

If a particular event should not be undertaken, then neither should any deacendent specializations of the event. These specializations may belong to plans having fault annotations that include other faults peculiar to the specialization, but these faults are of little consequence in light of a fault that indicates the plan should not be pursued. We term these special fault types to be *overriding faults*. The faults are of such importance that they override other faults in the fault annotation of a plan. We examine their use in the next chapter, but the basic idea is that we terminate clarification when we know that the user is intending to pursue an event that has an overriding fault associated with it, and we respond with a response that addresses the overriding fault.

3.6.7. Plan Recognition and Critiquing Methods

The clarification process is viewed to be a component of a much larger system. For example, the exact wording of clarification questions and a response to a user question may be assumed to be undertaken by some natural language management component, or they may be obtained from a database of "canned" queries and responses in the particular domain. Two components of vital importance to our particular clarification process are those of plan recognition and critiquing. Plan recognition allows us to provide cooperative responses to user questions. Critiquing the recognized plans allows us to respond to a user's question without necessarily determining the user's exact plan, thereby reducing interactive clarification dialogue.

The general clarification process, that we propose and describe in the next chapter, requires the following basic input:

- A set of possible plans that are annotated by some critiquing process. The set of plans is simply a list of plan labels which distinguish one plan from another. The fault annotation of any single plan may be a simple list of fault labels. Each possible plan has its own list.

- An event hierarchy that contains the events of the possible plans in the form of a graph, wherein each node represents a unique event and contains the following information:
  - A list of the disjunctive events (specializations) of the node. (These may be event labels or pointers to those nodes which are disjunctive children of the node).
  - A list of the conjunctive events (necessary components) of the node.
  - A list of those possible plans (plan labels) which contain (involve) the node.

Beyond these basic input requirements, the general clarification process that we propose does not rely upon any specific form of plan recognition or critiquing process.

Even though our clarification process does not assume any particular method of plan recognition or critiquing, we should address some of the issues involved when scaling up to a large system. With our hypothetical course-advisor, and a question about any university course, the possible plans a user might have may be formidable. Considering that there may be several faculties, each with many different departments with different programs of study, plan recognition and critiquing may present a significant problem. One solution that we propose is to first use clarification, before any plan recognition or critiquing takes place. That is, we first establish the top-level event or goal (to be an additional observation) with the use of clarification questions. For example, an initial dialogue of our course-advisor might be:

User: Can I take CMPUT 161?

System: Will you be entering the Faculty of
1. Arts
2. Science
3. Engineering
4. Business
5. Education

User reply: 2.

At this point, plan recognition may begin with two observations, i.e., the faculty and the computing course in the user's question. This assumes one vast plan library in the domain, but this does not have to be the case. If we had several plan libraries, say one for each faculty, the above reply would indicate that we use that plan library concerning the Faculty of Science and we give it the single observation of the event of the computing science course CMPUT 161. Plan recognition and critiquing could then begin and be restricted to the particular plan library. We could extend this idea (of clarification before plan recognition and critiquing) to more events at lower levels. In general, the more observations we supply to the plan recognition component, the fewer the number of recognized plans that will require critiquing. However, at some point, we may begin asking more questions than are necessary for providing a cooperative response. Had we used plan recognition and critiquing to begin with, we may not have required any clarification at all. But in dealing with very large numbers of plans, clarifying the top-level goals first, would seem to be a viable solution with respect to the efficiency of the plan recognition and critiquing procedures.

The next chapter outlines and describes a general clarification framework. Throughout, we keep the same format of the questions. That is, 'Yes/No' questions only involve single events, and menus are homogenous in that all menu items are single events that are all direct alternatives of the same event. Later we briefly examine issues related to relaxing these restrictions on the clarification questions.

Chapter 4. A General Clarification Framework

This chapter describes our proposal for a general clarification process. The first section provides the top level setting. The next section describes the data and data structures used by the clarification procedures. The remaining sections walk through different aspects of lower-level procedures providing algorithms in pseudo-code, examples, and explanations. We incorporate the strategies and concepts that address the limitations of the basic top-down approach, as described in the previous chapter. These different concepts are unified into a single, general clarification framework.

4.1. The Overall Setting

For convenience, we assume an advice-giving system that responds to a user's question with a cooperative response. Our top-level procedure for responding to a user's question is similar to that described in the previous chapter:

```
procedure GENERATE-RESPONSE(Query)
begin
        Check if Query is possible
        if Query fails then
                Output: "No, [Query] is not possible as [reason for failure]"
        else begin
                S ← PLAN-RECOGNITION(Query)
                S ← CRITIQUE(S)
                P ← Partition(S)
                if AMBIGUITY-MATTERS(P) then
                        S ← CLARIFICATION(S, P)
                RESPOND(Query, S)
        end
end
```

After the plans are critiqued, they are partitioned into different fault partitions according to the fault annotations of the possible plans. All plans with the same fault annotation (i.e., the same faults) will be in one particular partition. Ambiguity matters should the number of fault partitions be greater than one. The response answers the user's query with a direct answer, as well as addressing the fault annotation of the remaining plans for a cooperative response. When the RESPOND procedure is ılled, only one fault partition will remain. All remaining possible plans belong to that fa.... partition and have the same fault annotation.

4.2. The Data for Clarification

At the point that we begin our clarification, we have the required input of a set of annotated plans, an event hierarchy and the fault partitioning of the plans. The exact form of this data may vary. For clarity, we assume the following schemas:

1. A set of plans, where each plan consists of:

Plan Label	Faults	Partition Number

The *Faults* consist of a list of fault labels that make up the fault annotation of the plan and is determined by the critiquing process, but it is not used by our particular clarification process. The procedure that partitions the plans requires this data. Here, we only partition the plans once. Changes are made to these partitions, but the possible plans are never re-partitioned. The partition number designates which partition the plan belongs to and is obtained when the plans are partitioned. In addition to this basic data, we could possibly add a likelihood factor for the plan, as described in the previous chapter.

For convenience, we also keep the set of remaining possible plans as a simple list of plan labels and denote this list as **possible_plans**.

2. A set of **events**, where each event consists of:

Event	Specializations	Components	Plan Labels

This set of events is meant to represent the event hierarchy that results from plan recognition. The *Event* field may be the name of the event, or a unique event label to be used in place of the event name. All of the other fields are lists. The *Specializations* field is a list of events (or event labels) that are direct specializations (the disjuncts or alternatives) of the event. Similarly, the *Components* field is a list of the necessary components (the conjuncts or sub-branches) of the event. The last field is a list of those possible plans (plan labels) that involve or contain the particular event. In addition to this basic data, we might also include other fields to represent the lists of abstractions and uses of the event (i.e., the parents of the event). This would be necessary for a bottom-up approach (or possibly a middle-out tactic). Here, we always presume a top-down approach which never backs up or gets stuck in finding an event to ask the user about. If using the concept of overriding faults, then another field should be added, to contain the fault label(s) of the overriding fault(s).

3. A set of **partitions**, where each partition consists of:

Partition Number	Plan Labels

The *Plan Labels* field is a list of those possible plans (plan labels) which belong to the particular partition. All of these plans have the same fault annotation, i.e., the same set of faults. As we clarify, this list may be reduced. When the list is empty, we consider the partition to be eliminated.

For convenience, we maintain a list of remaining partitions (a list of partition numbers), and denote this list to be **remaining_partitions**. We also maintain the number of remaining partitions and designate this variable to be **n** partitions. Thus, ambiguity matters when (**n** > 1) is true.

Applying this particular schematic form to the recognized plans and event hierarchy of figure 3.7 on page 43, we would obtain the following result (example 4.1):

The possible plans are:
(p1, [f0], 1)
(p2, [f0], 1)
(p3, [f1], 2)
(p4, [f0], 1)
(p5, [f0], 1)
(p6, [f1], 2)

(f0 indicates faultless, and f1 indicates a fault concerning a vegetarian guest)

The plans correspond to the following:

label	*plan*	*fault annotation*
p1:	(red wine with fettucini marinara):	*faultless*
p2:	(red wine with spaghetti marinara):	*faultless*
p3:	(red wine with chicken marinara):	*vegetarian guest*
p4:	(white wine with fettucini marinara):	*faultless*
p5:	(white wine with spaghetti marinara):	*faultless*
p6:	(white wine with chicken marinara):	*vegetarian guest*

possible_plans = [p1,p2,p3,p4,p5,p6]

The events are:
(end, [entertain guest], [], [p1,p2,p3,p4,p5,p6])
(entertain guest, [], [choose wine, prepare meal], [p1,p2,p3,p4,p5,p6])

(choose wine, [choose red wine, choose white wine], [], [p1,p2,p3,p4,p5,p6])
(choose red wine, [choose French wine], [], [p1,p2,p3])
(choose white wine, [choose French wine], [], [p4,p5,p6])
(choose French wine, [], [], [p1,p2,p3,p4,p5,p6])

(prepare meal, [make pasta dish, make meat dish], [], [p1,p2,p3,p4,p5,p6])
(make pasta dish, [make fettucini marinara, make spaghetti marinara], [],
[p1,p2,p4,p5])
(make meat dish, [make chicken marinara], [], [p3,p6])
(make fettucini marinara, [], [make marinara sauce], [p1,p4])
(make spaghetti marinara, [], [make marinara sauce], [p2,p5])
(make chicken marinara, [], [make marinara sauce], [p3,p6])
(make marinara sauce, [], [], [p1,p2,p3,p4,p5,p6])

The partitions are:
(1, [p1,p2,p4,p5]) *(the faultless partition)*
(2, [p3,p6]) *(the vegetarian partition)*

remaining_partitions = [1,2]

n = 2 *(there are only two fault partitions)*

Example 4.1 The data for the event hierarchy of figure 3.7

Once we begin clarification we only make changes to the data with respect to:
a. The possible plans. Each clarification question reduces the set of (remaining) possible plans.
b. The plans associated with a partition. These plans may be reduced in number through clarification questions. If these plans are all eliminated, then so is the partition.
c. The number of partitions may be reduced.

After each clarification question (and reply) we:
1. adjust the possible plans, removing eliminated plans.
2. remove eliminated plans from the partitions containing them.
3. remove partitions with empty sets of plans.
4. check ambiguity. It will matter if there is more than one partition. If this is the case we keep clarifying, otherwise we respond.

4.3. The Clarification Procedure

There are two cases we consider for clarification:
1. No menus, and only 'Yes/No' type questions.
2. Menus and 'Yes/No' type questions.

In either case, we are allowed to use key events ('Yes/No' questions), when deemed to be appropriate. Otherwise, we clarify the possible set of plans the user may be following by traversing the hierarchy in a top-down manner. At each branchpoint with specializations, we ask the user about the possible alternative events directly below that branchpoint. We do this with either one menu question, in which each menu item is an alternative, or we ask about each alternative with a 'Yes/No' question. If doing the latter, and only one alternative remains, we do not ask it, but assume it is the alternative that the user is following. Throughout the clarification procedure, we are not allowed to go back up the hierarchy to some previous branchpoint and we are not allowed to get "stuck" (and not have another question to ask which will resolve the ambiguity).

In traversing the hierarchy, we will always have one, and only one, current branchpoint. At the start of clarification, this will be the *end* event, which can only have specializations (events that are direct disjuncts, or alternatives of the *end* event) each of which corresponds to a possible goal of the possible plans. Whenever we are at an event that is the current branchpoint (CB), there are several possible cases, in determining the next branchpoint. These cases are the following:

a. The CB has only one child. In this case the event that is the CB is not really a branchpoint in the sense that it does not branch. It does not matter if the only child is a specialization (disjunct) or a necessary component (conjunct). In either case, we make the child the new CB and continue from there.

b. The CB only has disjunctive children (alternatives or specializations of the CB). This will be the case at the very start when the *end* event is the CB. We either clarify these alternatives or we skip ahead to some other branchpoint lower in the hierarchy, if that possibility exists and it is deemed "safe" to do so. If we do not skip, then one of the alternatives will become the next CB.

c. The CB only has conjunctive children (necessary component events required for a single plan; these are not alternatives representing different plans). In this case, each conjunctive child is a possible future branchpoint. We must record

the conjunctive children, since we may have to come back to them later. To do this, we maintain a *stack* structure of branch points which we term to be the *branchpoint stack* (BPS). Initially we set the *end* event to be the CB and create a new empty stack. In this case, of a CB having only conjunctive children, we push each child onto the stack. Since there are no alternative children, our next current branchpoint will be the top of the stack.

d. The CB has both conjunctive and disjunctive children. In this case, we push each conjunctive child onto the BPS and we deal with the alternatives as in case (a) or (b) above, depending on the number of alternatives.

e. The CB has no children. It is a leaf event in the event hierarchy. In this case, we pop the BPS, and set the result to be the CB. If the BPS is empty and we still have more than one remaining partition, then there is an error.

In clarification, we are attempting to determine which alternative events (plans) the user is following. Component events (or conjuncts) do not distinguish different plans, and we do not ask about them, although there may be a case for referring to them in order to maintain context and coherence with the user. In traversing the hierarchy, we are looking for those branch points that have more than one alternative, and we ask about these alternatives. In essence, we always want to find the next such branchpoint and have it as the CB.

The general clarification procedure is as follows (the line regarding key events will be explained in the section dealing with key events, and procedure CLARIFY is given later in this section):

```
procedure CLARIFICATION(S,P)
/* S = possible plans , P = partitions ; both are actually global to all sub-procedures*/
begin
        CB ← end event
        BPS ← stack (new)
        if KEY_EVENTS_POSSIBLE(CB) then KEY_EVENTS

        while AMBIGUITY-MATTERS(P) do
          begin
            Push each conjunctive child of CB onto BPS
            if CB has no alternatives then CB ← BPS (pop)
            else
                if CB has one alternative then CB ← one alternative
                else CB ← CLARIFY(CB)
          end

        return (plans of P).  /* where P is the remaining partition */
end
```

Whenever the current branch point has more than one alternative, we ask a clarification question which provides the next branch point. After any clarification question, the remaining plans and partitions are adjusted, and we check for ambiguity mattering, before we deal with the next branch point. Ambiguity will not matter if there is

only one remaining partition, in which case we return the remaining plans, which all belong to the one partition.

To illustrate the above algorithm, we can step through example 4.1. To recap, there is a user question with a multiple observation concerning French wine and making marinara sauce. Plan recognition, critiquing, and partitioning of the plans will give us the data of example 4.1. In this case, assume that we do not account for skipping branch points, using key events, or overriding faults. In essence, we just illustrate the basic top-down approach of the previous chapter, except we are now dealing with complex plans. The reader may more easily follow the steps by referring to figure 3.7.

Step 1: (CB = end)

Initially, the current branch point (CB) is set to be the *end* event and our branchpoint stack (BPS) is empty. Since there are two partitions, ambiguity matters. End events never have conjuncts, so the BPS remains empty. Since this end event has only one alternative of *entertain guest*, we set CB to be that one alternative. This completes one pass through the while loop. The only change is that CB is now the event of *entertain guest*, rather than the *end* event.

Step 2: (CB = entertain guest)

The event of *entertain guest* has two conjunctive children. In this case assume we first push *prepare meal* onto the BPS and we then push *choose wine* onto the BPS. Since entertain guest has no alternatives, we pop the stack and set CB to the result. CB is now the event of *choose wine* and the BPS contains only one sub-branch of *prepare meal*. This completes the second pass through the loop.

Step 3: (CB = choose wine)

The event of *choose wine* has no conjuncts, so the BPS does not change. However, the event does have two alternatives of *choose red wine* and *choose white wine*. The event is a disjunctive branch point. The event is passed to the CLARIFY procedure, which will ask the user clarification questions about the disjuncts of *choose wine*. Assuming the result indicates a white wine, the plans and partitions are adjusted to this event. We keep plans involving white wine and eliminate plans involving red wine. This gives us the following result:

- possible_plans = [p4,p5,p6]
- the partitions are (1, [p4,p5]) and (2, [p6])
- remaining_partitions = [1,2] *(no change)*
- n = 2 *(no change)*

The CLARIFY procedure returns that branch point which results from clarification questions, and we set CB to this result. In this case CB is the event of *choose white wine*. Ambiguity still matters, since we still have two partitions.

Step 4: (CB = choose white wine)

The event of *choose white wine* has no conjuncts and only one alternative of *choose French wine*. CB is set to this one alternative and no other changes occur.

Step 5: (CB = choose French wine)

The event of *choose French wine* is a leaf event. Since there are no conjuncts, the branchpoint stack does not change. Since the event has no alternatives, we pop the stack and set CB to be the result. At this point, CB is the event of *prepare meal* and BPS is empty. This completes another pass through the loop.

Step 6: (CB = prepare meal)

The event of *prepare meal* has two alternatives of *make pasta dish* and *make meat dish*, so we again pass the event to the CLARIFY procedure and ask the user clarification questions. Assume the first question to be about making a pasta dish and the user reply to be 'Yes'. We keep all pasta dish plans and remove all plans involving a meat dish. This gives us the following result:

- possible_plans = [p4,p5]
- the partitions are (1, [p4,p5])
- remaining_partitions = [1]
- n = 1

The remaining meat dish plan is eliminated, leaving the partition concerning a vegetarian guest with an empty set of plans. The partition is therefore eliminated, leaving only one other remaining partition containing two faultless plans. The CB is set to the result of the CLARIFY procedure which is the event of *make pasta dish*.

Step 7: (CB = make pasta dish)

At this point the condition of the while loop fails. Since there is only one remaining partition, ambiguity no longer matters. The two remaining plans (p4 and p5) are returned. The system can then provide a cooperative response in light of these two plans. In this case, both are faultless.

The CLARIFY procedure depends upon the particular mode the system is using. Here, we assume three possible cases that can be toggled.

```
procedure CLARIFY(CB)
begin

    Case mode is
        menus: branchpoint ← CLARIFY_MENUS(CB);
        likelihoods: branchpoint ← CLARIFY_LIKELIHOOD(CB)
        default: branchpoint ← CLARIFY_BASIC(CB).

    return (branchpoint)
end
```

Here, the CLARIFY procedure is only meant to distinguish which mode the system is using, and hence the exact clarification method to be used. If we are using menus, we use CLARIFY_MENUS. If using likelihood factors, we use CLARIFY_LIKELIHOOD. Without using menus or likelihoods, clarification is limited to 'Yes/No' questions about events that are assumed to be equally likely. This is the basic situation and is termed CLARIFY_BASIC. With all three procedures, the argument parameter (the current branch point) is a branch point with more than one alternative. The branch point that is returned will either be one of these alternatives, or some other branch point in the case that we are able to skip branch points. The next section describes the CLARIFY_BASIC procedure. The other two procedures basically involve certain changes to the CLARIFY_BASIC procedure.

4.4. Clarify Procedure with no Menus and no Likelihoods (CLARIFY_BASIC)

The situation is that we have a current branchpoint (CB) with more than one alternative (specialization) and n fault partitions, with (n > 1) since ambiguity still matters.

Our overall goal is to provide a response to the user's question by asking the user as few clarification questions as possible. We attempt to achieve this goal by reducing the number of fault partitions as quickly as possible. The problem is that of selecting events to query the user about which have the most effect on the number of remaining partitions. Conversely, we want to avoid asking about events that have no effect upon the number of remaining partitions. There are three possible avenues we explore:

1. We can ask the alternatives of the CB. This approach maintains the current context and coherency of the dialogue. We term this approach to be the *hierarchical* approach.

2. It may be possible to skip ahead to another branch point where all the plans having CB also have that particular branch point. This is termed to be *branch point skipping*, and is applied to diamond-type configurations and the sub-branches of complex plans.

3. We can select an event from the hierarchy that is unique to a certain partition, but is an event of every plan of that partition. We term such events to be key events. Asking the user about a key event will either eliminate one fault partition or establish that partition as the only possible partition. We check for the possibility of using key events before the CLARIFICATION procedure. After any clarification question, we also check to see if key events may be used. Once they are used, we keep using them until only one partition remains.

We discuss branch point skipping and key events in detail in later sections. In this section we are concerned with asking the possible alternatives of the CB. In this case, the likelihood of plans is not considered. That is, we do not have any data on the likelihood of one plan over another, or the application is such that plans are considered to be equally likely. Also, the application is such that a menu question is not feasible, as with an automated telephone system. The problem, then, is that of selecting that alternative of the current branch point to ask first. We propose to determine this by looking ahead to the number of remaining partitions for each alternative, if the user should reply with either a 'Yes' or 'No' answer to that alternative. Therefore, we determine the 'Yes' and 'No' *result* for each alternative of CB (in terms of the number of remaining partitions), before asking the user about any particular alternative.

A 'Yes' reply to an event means that the user will be doing that event. We would reduce the remaining possible plans to those plans that have that particular event. Only partitions that contain such plans will remain, and any partition that does not contain a plan having that event will be eliminated. Since we know which plans involve an alternative event, and we know which partition any plan belongs to, we can determine the number of fault partitions that contain a certain event. This number of fault partitions will be the 'yes' result of an event.

If each 'yes' result of every alternative of CB does not reduce the number of partitions, then we have a special case. We can show that each 'no' result will also not reduce the number of partitions. That is, if each 'yes' result of every alternative is n partitions, then it must be the case that every 'no' result will be n partitions. The only way in which a 'No' reply to a particular event will eliminate a partition, is if every plan in that partition has that particular event. If one, and only one, of the possible alternatives of the current branch point is present in a certain partition, then a 'No' reply eliminates that partition.

Given n partitions ($n > 1$), and x alternatives of the current branch point ($x > 1$), then a 'yes' result of n partitions for each of the x alternatives, will mean that each alternative is present in every partition. A 'Yes' reply to any particular alternative will leave us with the same number of partitions, since that alternative is in every partition. Since we have more than one alternative ($x > 1$), then every partition must contain at least x alternatives. In this case, a 'No' reply to any particular alternative will not eliminate any partitions, since every partition must contain at least one plan having a different alternative. Therefore, should all 'yes' results be n partitions, all 'no' results must also be n partitions (no partition can be eliminated with a 'No' reply). For example, if we have two alternatives of red wine and white wine and each type of wine is involved in some plan of each partition, then a 'Yes' or 'No' reply to a question about either wine will leave the same number of partitions.

Should every alternative have a 'yes' result of n partitions, then we know that every alternative will have a 'no' result of n partitions. This not only saves us from calculating the 'no' results, but it also means that we may be able to skip branch points. We are able to skip branch points if all intervening events (events between the branch points) are not required to affect a response. Every skippable branch point will have this initial situation, that is, a current branch point (CB) with n partitions and more than one alternative, with each alternative having a 'yes' result of n partitions, and a 'no' result of n partitions. This situation does not mean that we can skip to another branchpoint; only that it may be possible. Whenever this situation arises, we check for the possibility of skipping branch points.

If the 'yes' results are not all n partitions, then we cannot skip branch points. At least one alternative reduces the number of remaining partitions, and therefore, at least one alternative makes a difference with respect to the response. We must clarify the exact alternative the user is pursuing. Since the remaining alternatives (plans) are equal with respect to likelihood, any alternative we select for a query to the user will have a $(1/x)$ probability of being correct (given x alternatives). To determine which alternative to ask first, we also determine the 'no' result for each alternative.

To determine the 'no' result for each alternative, we need to determine the number of partitions it is exclusive to. The plans of such a partition must all be plans having the alternative event. The number of such partitions, for a particular alternative, will be the number of partitions eliminated on a 'No' reply to that alternative. The 'no' result for that alternative will be n minus this number, giving us the number of remaining partitions on a 'No' reply to the event.

Once we have the 'yes' results we can make certain decisions:

1. If the 'yes' results are all n, we check for branch point skipping. If we can skip we do so. If we cannot, then we ask the alternatives in any order. Since the 'no' results are also all n partitions, none of the alternatives will affect the number of remaining partitions.

2. If the 'yes' results are not all n, then we determine the 'no' result for each alternative. We base our selection on the 'yes' and 'no' results in the following manner:
   Assume that all the alternatives are equally likely with probability $1/x$.
   Determine the expected number of remaining partitions for each alternative; $1/x$('yes' result) + $(1 - 1/x)$('no' result).
   Select the alternative with the minimum expected number of remaining fault partitions. Ties are broken randomly.

In this way we will, on average, reduce the number of fault partitions more quickly than by an arbitrary choice.

Assuming there are more than two alternatives, and we select one of these to ask the user, we must account for changes in case of a 'no' reply to that event, since it may affect the other remaining alternatives. The 'yes' results of the remaining alternatives will not change and we can simply reuse them. We need to recalculate the 'no' results for the remaining alternatives, since some plans may have been eliminated from partitions containing them, now making the alternative exclusive to such partitions when it was formerly not exclusive to those partitions.

The general algorithm for the CLARIFY_BASIC procedure is given below:

```
procedure CLARIFY_BASIC(CB)
begin
        next_branchpoint ← nil

        for each alternative of CB, find its 'yes' result

        if all 'yes' results are n partitions then
            begin
            NB ← BRANCH_POINT_SKIP(CB)
            if NB is not nil then next_branchpoint ← NB
            else next_branchpoint ← CLARIFY_ALTERNATIVES(CB).
            end

        else next_branchpoint ← CLARIFY_ALTERNATIVES(CB)

        return (next_branchpoint).
end
```

```
procedure CLARIFY_ALTERNATIVES(CB)
begin
        alt_events ← set of alternatives of CB
        branchpoint ← nil

        while ((branchpoint is nil) and (AMBIGUITY-MATTERS(P))) do
         begin
               for each alternative of alt_events, find its 'no' result
               selection ← one of alt_events based on 'yes/no' results

               Ask user about selection

               if reply is 'Yes' then
                  branchpoint ← selection
                  possible_plans ← selection's plans ∩ possible_plans
                  ADJUST-PLANS(selection)

               else  /* the reply is 'No' */
                  alt_events ← alt_events – selection.
                  if alt_events has only one Event then
                   branchpoint ← one Event
                   possible_plans ← one Event's plans ∩ possible_plans
                   ADJUST-PLANS(one Event)
                  else
                   possible_plans ← possible_plans – selection's plans
                   ADJUST-PLANS(nil)
         end

        return (branchpoint)
end

procedure ADJUST-PLANS(new-branchpoint)
begin
        for each fault partition do
          begin
               Remove eliminated plans from partition's plans
               if partition has no plans then remove partition  /* i.e., n ← (n-1) */
          end

        if AMBIGUITY-MATTERS(P) then  /* n > 1 */
               if KEY_EVENTS_POSSIBLE(new-branchpoint)
               then KEY_EVENTS.

        return
end
```

To illustrate the CLARIFY_BASIC procedure, let us assume our course-advisor domain. Assume that the user's question pertains to a particular mathematics course, Calculus 102, and that the event hierarchy that results from plan recognition is that of figure 3.3 on page 34, with the current branch point being the event of *Area of Science Concentration*. That is, we have already established that the user intends to obtain a BSc. degree. For this example, we also assume that key events are not usable. Since the current branch point has 4 alternatives, and we are not using any likelihood factors or menus, the CLARIFY_BASIC procedure is used for clarification questions. At this point, let us assume the following:

- The user's plan involves a specialization program in the department of Computing Science (i.e., in the area of Mathematical Sciences).
- There are 17 remaining possible plans.
- The 17 possible plans are partitioned into 8 remaining partitions. That is, the critiquing procedure has determined fault annotations, and the partitioning of the remaining 17 possible plans (according to their fault annotations) corresponds to 8 different fault partitions. Since there is more than one partition, ambiguity matters.
- The 8 partitions are simply enumerated and depicted in figure 4.1 (the type of program, indicated by (H) for Honors and (S) for Specialization, and the specific department is used to specify a certain plan, rather than the plan label; each event has its direct abstraction area denoted in brackets):

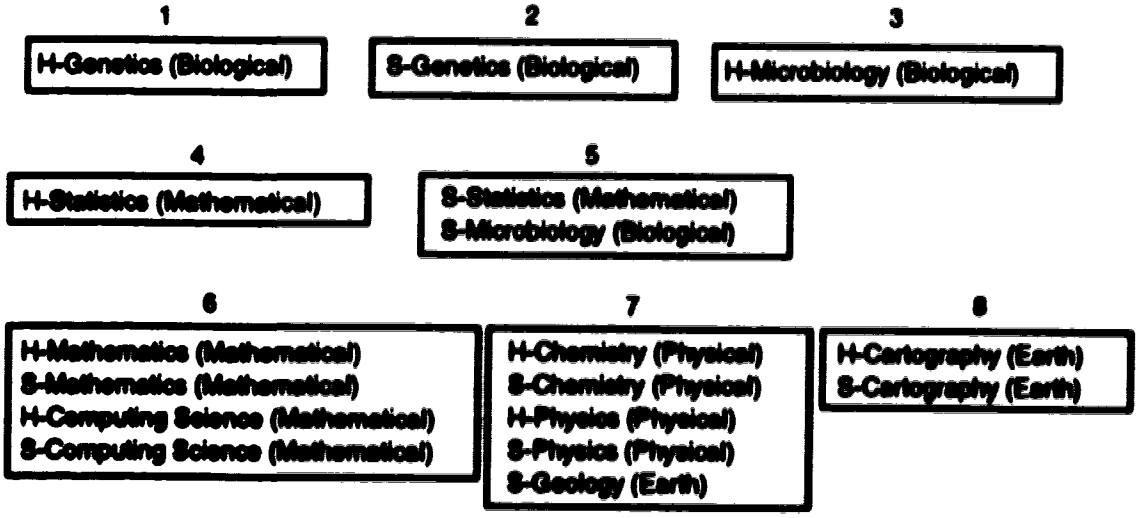

Figure 4.1 The partitioning of Science plans for course Calculus 102.

In this example, the CLARIFY_BASIC procedure is given the branch point event of *Area of Science Concentration*, which has four alternatives. With this particular situation, we have 8 remaining fault partitions (n = 8). In this case, the 'yes' result of each alternative is:

Physical Sciences:	1
Mathematical Sciences:	3
Earth Sciences:	2
Biological Sciences	4

Since all alternatives do not have a 'yes' result of n partitions (n = 8), we cannot skip branch points. Therefore, we must clarify the alternatives with the CLARIFY_ALTERNATIVES procedure. We determine the 'no' result of each alternative. *Throughout this example the alternative with the minimal 'no' result is the alternative with the minimal expected number of remaining fault partitions.* The 'no' results are:

Physical Sciences:	8 (i.e., a 'No' reply leaves 8 partitions)
Mathematical Sciences:	6
Earth Sciences:	7
Biological Sciences	5

The minimum 'no' result, is that result for the alternative of *Biological Sciences* which has the minimal expected outcome (i.e., 1/4(4) + 3/4(5) = 19/4). Asking this event will leave us with 5 remaining fault partitions on a 'No' reply, and 4 remaining fault partitions on a 'Yes' reply. According to our proposed rules, we would select the event of *Biological Sciences* to first ask the user. With this example, the user's reply would be 'No'. As a result of this reply, the possible plans are reduced to those plans that do not involve the event of *Biological Sciences*. When adjusting the plans, three partitions are eliminated, and one partition has a "Biological plan" removed. At this point the remaining partitions are represented by figure 4.2.

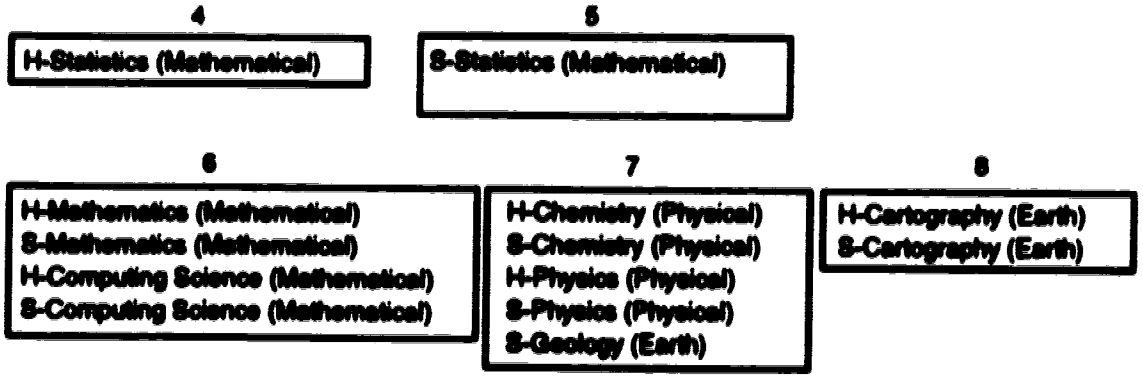

Figure 4.2 Remaining Partitions without Biological Science Plans

After adjusting the plans, we are left with 5 partitions (n = 5). Since more than one alternative remains, another branch point has not yet been established and ambiguity still matters. Another alternative must be asked. The 'yes' result of the remaining alternatives does not change, but the 'no' results are now different. The results are:

	Yes	No
Physical Sciences:	1	5
Mathematical Sciences:	3	2
Earth Sciences:	2	4

Based on the new 'no' results, the next alternative to ask about is that of *Mathematical Sciences*. A 'No' reply to this event will leave two partitions. In this case, the user's reply will be 'Yes'. The possible plans are now those plans that involve the event of *Mathematical Sciences*. In the algorithm, we determine the set of new possible plans by the intersection of the set of the event's plans with the set of the previous possible plans. Since we do not change the data concerning events, it may be the case that the event's plans include plans which have already been eliminated, and simply acting

the possible plans to the event's plans would be incorrect. After adjusting the plans, three partitions remain, as shown in figure 4.3.

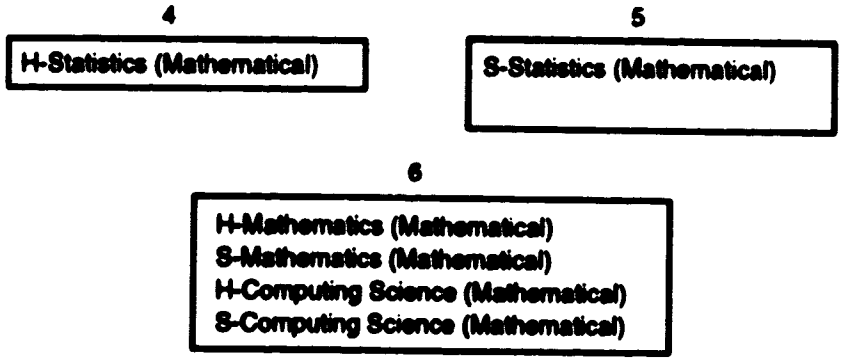

Figure 4.3 Partitions for plans with Mathematical Sciences

The CLARIFY_BASIC procedure returns the next branch point event of Mathematical Sciences. This event has no conjuncts, but three alternatives for the departments of Mathematics, Statistics, and Computing Science. As a result, we would again use the CLARIFY_BASIC procedure with the new current branch point of *Mathematical Sciences*, and we again need to clarify the alternatives of this event. The 'yes' and 'no' results for these alternatives are:

	Yes	No
Mathematics	1	3
Statistics	2	1
Computing Science	1	3

Using the selection rules, we first ask the user about entering the department of Statistics (rather than Mathematics or Computing Science). Since the user is planning to enter Computing Science, the reply is 'No'. After adjusting the plans, we are left with one partition (partition number 6) with four remaining possible plans. At this point, ambiguity no longer matters, and a response is provided for the plans of partition number 6, e.g., "You may take the course Calculus 102, but Calculus 104 is strongly recommended by the departments of Mathematics and Computing Science".

4.5. Clarify Procedure with Likelihoods (CLARIFY_LIKELIHOOD)

The procedures are essentially the same as that for the CLARIFY_BASIC algorithms described in the previous section. The only difference is that disjunctive events of a branch point are no longer equally likely, and the selection process needs to be adjusted.

The exact mechanism of determining that a particular plan is more likely than some other plan, is not of concern to us here. Many different methods could be used. One was described in the previous chapter on the basic top-down approach. Another possibility is that of assigning probabilities to alternative events as was done by Calistri-Yeh. Whatever mechanism is used, we assume it will allow us to prefer one alternative of a branch point over another alternative of the same branch point. We can replace the selection process based on 'yes' and 'no' results with a procedure that heuristically orders the alternatives of a branch point. The clarification questions then use this ordering until the next branch point is determined or ambiguity no longer matters.

4.6. Clarify Procedure with Menus (CLARIFY_MENUS)

The situation is that we have a current branchpoint (CB) with more than one alternative (specialization) and n fault partitions with n > 1, since ambiguity still matters.

The alternatives are presented to the user in some type of menu format with each alternative being a separate menu item. This format will have a preamble dialogue suitable to the context of the possible alternatives. The user's reply will be the selection of one menu item, which means that the user is intending to follow or do that particular alternative.

The procedure is similar to the previous CLARIFY_BASIC procedure. The basic difference is that we never deal with the 'no' results of the alternatives, or their likelihood. The algorithm would be the following:

```
procedure CLARIFY_MENUS(CB)
begin
        for each alternative of CB, find its 'yes' result

        if all 'yes' results are n partitions then
            begin
                NB ← BRANCH_POINT_SKIP(CB)
                if NB is not nil then
                  begin
                        next_branchpoint ← NB
                        return (next_branchpoint)
                  end
            end

        Construct and show menu
        selection ← user's menu choice of alternatives of CB
        next_branchpoint ← selection
        possible_plans ← selection's plans ∩ possible_plans

        ADJUST-PLANS(selection)
        return (next_branchpoint)
end
```

4.7. Skipping Branch Points

For the purpose of reducing the number of clarification questions required to provide a response, it is advantageous to skip branch points if possible. The possibility of skipping branch points occurs with a certain situation. The situation is that we have a current branchpoint (CB) with more than one alternative (specialization) and n fault partitions with n > 1, since ambiguity still matters. Each alternative of CB has a 'yes' result of n partitions, and consequently a 'no' result of n partitions. This means that each remaining partition has every alternative in some plan of that partition. If we were to ask any alternative, we will be left with n partitions (and n possible responses). This situation indicates that we may be able to skip branch points. If the 'yes' results were not all n

partitions, then we cannot skip branch points, since some alternative of the current branch point is required to determine the eventual response.

There are two possibilities with respect to the branch point we can skip to. If the paths descending from the current branch point all converge to some single event lower in the event hierarchy (and that event is not a leaf event), then we have a *diamond* configuration, with the *diamond top* being the current branch point and the *diamond bottom* being that event at which the paths converge. In this case, all remaining plans have the events of the diamond top and the diamond bottom, and the branch point we skip to will be the event that is the diamond bottom. Should the paths descending from the current branch point not converge to a single event, but end in leaf events, then the branch point we skip to will be the sub-branch event at the top of the branchpoint stack (BPS). We can also view this situation as being similar to a diamond configuration. Figure 4.4 illustrates this for our cooking domain.

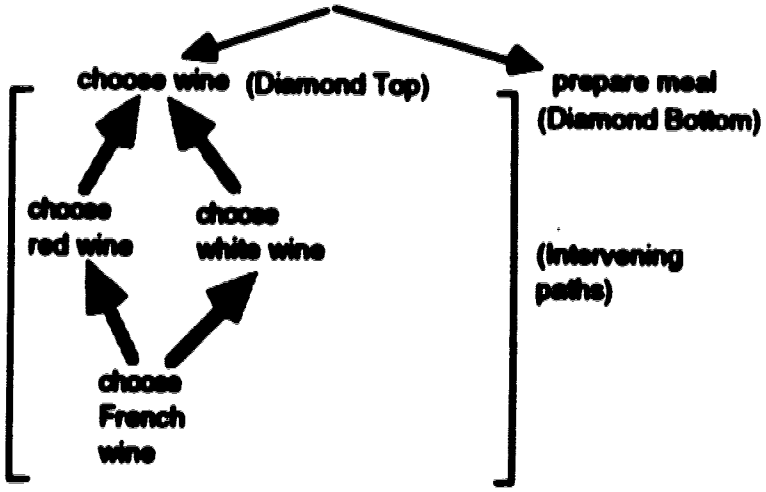

Figure 4.4 The choose wine sub-branch, viewed as a diamond situation

We can only skip branch points if all *intervening events* are not required to resolve ambiguity to the point that it no longer matters. In the case of diamonds, the intervening events are those disjunctive events, of the event hierarchy, that are between the current branch point and the event of convergence. In the second case, the intervening events are those events from the current branch point to leaf events (that is, the branchpoint stack has the *diamond bottom*). No matter which intervening events we may ask, we must eventually come to the branch point that we could have skipped to. All plans having the current branch point (all remaining plans) must also have the branch point that we skip to, whether this is a converging event or the sub-branch at the top of the BPS.

Another viewpoint to the "safety" of branch point skipping is that we can skip branch points if the ambiguity of the intervening paths (or "sub-plans") does not matter to the response. If we can determine that all intervening paths are all faultless, or that all have the same fault(s), then their ambiguity does not matter to the response. The user's plan must follow one of these intervening paths. There are only n fault partitions and n possible responses. The intervening paths must be located in these n partitions, but if all intervening paths have the same annotation, then the n different fault annotations are not determined by any of the intervening paths, but by alternative paths occurring after the branch point we skip to.

Although the above viewpoint may be simpler, the determination of faults for a path (certain events that form a portion of a plan) presents certain problems. We did experiment with the possibility of only having faults associated with single events. This would allow us to ascertain the faults of a path of events. However, it also imposes restrictions on the critiquing process and the configuration of the plan library. That is, we can see no reason why a fault cannot be the result of a certain combination of events in a plan, rather than resulting from a single event. For example, red wine by itself may be faultless, and chicken marinara by itself may be faultless, but red wine with chicken marinara may not be considered to be a good combination and should have a fault. We cannot assign the fault to either single event (in our given cooking library), but we might change the plan library to account for this fault. In light of this, we do not assume that faults are associated with individual events. Rather, we determine if we can end clarification by only using the events that occur below or "after" the branch point that we skip to. This approach is more general and does not impose any restrictions on the critiquing process, nor any rules with respect to the design of the hierarchy of plan libraries.

In order to determine if we can skip branch points, we must first determine the presence of a diamond bottom. There are two different cases for a bottom. In the one case, all remaining plans may converge at some single event. If this occurs, that event is our diamond bottom. In the second case, all remaining plans may not converge, but our branchpoint stack is not empty. In this case the diamond bottom, or rather the branchpoint we skip to, will be that event which is at the top of the stack. In the first case, the intervening events are those disjunctive events on paths from CB to the event that is the diamond bottom. In determining the intervening events, we only consider "simple" paths from CB to the event of convergence. We do not consider any intervening branch points that have sub-branches (i.e., we only consider events that have a single child, or events that only have alternatives as children). Should any path not be simple, we do not skip branch points. The reason for this is that it is unlikely that such conjunctive paths will all converge to a single event, and we may inadvertently bypass a possible future branch point. This is not the case for non-diamond situations, since all remaining paths must implicitly converge to the branch point at the top of the stack. In this second case, the intervening events are those disjunctive events from CB to some leaf event, including that leaf event. If we do not have any diamond bottom (a branch point to skip to) we cannot skip. If we do get a diamond bottom, we determine if we can safely skip to that branch point.

The BRANCH_POINT_SKIP procedure returns the branch point to skip to or a "nil" result indicating that the current branch point cannot be skipped.

```
procedure BRANCH_POINT_SKIP(CB)
begin
        bottom ← FIND_DIAMOND_BOTTOM(CB)
        if bottom is nil then return (nil)

        intervening ← FIND_INTERVENING_EVENTS(CB, bottom)
        if intervening is nil then return (nil)

        safety ← SAFE_SKIP(CB, intervening, bottom)

        if safety is true then
                        if bottom = BPS(top) then return (BPS(pop))
                        else return (bottom)
        else return (nil)
end
```

In the FIND_DIAMOND_BOTTOM procedure, a single path from CB can be traced downward in the hierarchy. If during this trace, we come upon an event, that is not a leaf event, at which all remaining plans converge, that event is returned as the diamond bottom. If this does not occur, then we eventually come upon some leaf event. In this case, if the branchpoint stack is not empty, the procedure returns the top of the stack as the branch point to skip to. Otherwise, the BPS is empty and there is no diamond bottom, so the procedure would return a nil result, indicating that we cannot skip branch points.

The FIND_INTERVENING_EVENTS procedure returns a set of intervening disjunctive events between CB and the bottom, or a nil result. The procedure can determine which case applies according to the bottom. That is, if the bottom is the event at the top of the BPS, then we are in the second case. Otherwise, we are in the first case (a diamond configuration) and we determine the intervening disjunctive events between CB and the converging event of the bottom. If during this determination for the first case, we come upon events with sub-branches (i.e., non-simple paths), then we should not skip. If this occurs, we immediately return a nil result.

4.7.1. Determining the Safety of Skipping Branch Points

The last procedure of SAFE_SKIP determines if we can resolve the ambiguity to the point that it no longer matters by only asking about those events that occur "after" the bottom. That is, if we can reduce the remaining partitions down to one partition without resorting to any intervening events, then it is safe to skip branch points from CB to the bottom. To determine this, we establish a set of *common-exclusive* events for each fault partition. If every partition has a valid and non-empty common-exclusive set, then we can skip branch points, and a 'True' result is returned. Otherwise, a 'False' result is returned by the SAFE_SKIP procedure.

Each event of a common-exclusive set is a disjunctive event that is common to all of the plans of a particular partition. The set does not include events that have already been established. Established events are those from the *end* event to CB, plus the bottom

event (all remaining plans share these events). Furthermore, the set cannot include any intervening events. Thus, every event of a common-exclusive set is a disjunctive event that is a descendant of the bottom or occurs after the bottom. In order to be valid, the set must be exclusive to the plans of its particular partition. That is, plans in other partitions cannot hold the same set of events (certain events may be shared by plans from different partitions, but the set as a whole cannot be). Each event of a valid set is common to all plans of a particular partition and the entire set is exclusive to the plans of that partition. Each such set is then unique to the plans of the partition for that set and only includes disjunctive events occurring after the bottom. Note that if a particular valid set has only one event, plans in other partitions cannot hold this event. In essence, that event is a key event.

If every partition has a valid and non-empty common-exclusive set, then we can safely skip branch points from CB to the bottom. In resuming the hierarchical approach from the bottom, we eventually substantiate the last member of the common-exclusive set of the partition containing the user's plan. When this occurs, only one partition will remain, since the entire set has been substantiated and is unique to one and only one partition. We could conceivably ask a single question concerning the common-exclusive events of each partition, such as "Are you doing *X* and *Y* and ...", where each letter is a common-exclusive event. A reply of 'Yes' would substantiate only those plans of one partition.

To show that this criterion, of every partition having a valid and non-empty set of common-exclusive events, allows us to safely skip branch points, we can assume a situation in which we do skip branch points and get "stuck" because we did not clarify the intervening events.

Assume we skip branch points from the current branch point of CB to Branchpoint X and we get stuck with no more questions to ask in the hierarchy (ambiguity still matters). Then there are at least two remaining fault partitions, one of which contains the user's plan. All remaining plans share certain disjunctive events. These are the disjunctive events from the end event to that branch point event that was skipped (CB) as well as the disjunctive events from Branchpoint X to some leaf disjunctive event in the hierarchy. That is, the only difference between the remaining plans is that of the intervening disjunctive events between CB and Branchpoint X. If we label the common exclusive sets of the partitions, and assume two partitions (the argument easily extends to more partitions), figure 4.5 illustrates the situation.

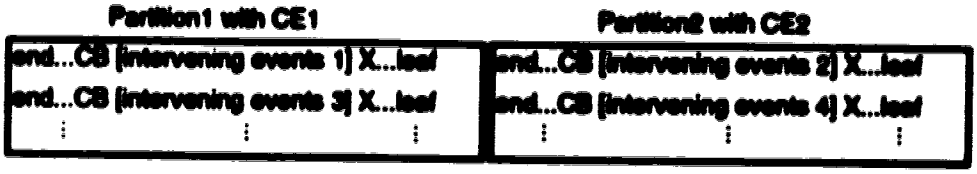

Figure 4.5 The "stuck" situation

In figure 4.5, those events denoted by "end...CB" are the same for every remaining plan. Similarly, those events denoted by "X...leaf" are the same for every remaining plan. Only the intervening events distinguish one plan from another and these were events not clarified (or substantiated) because they were skipped over. Both common-exclusive sets of CE1 and CE2 were established as valid and non-empty, in allowing the skip from CB to X. This means that CE1 ≠ CE2. Only those plans in

Partition1 contain the particular combination of events contained in CE1, so CE1 is exclusive to Partition1. Furthermore, neither common-exclusive set can be a proper subset of the other. The events in either set must come from those events from X to leaf, and these are the same for all remaining plans. For example, CE1 cannot be a subset of CE2, because plans in Partition2 would contain all events of CE1 making CE1 not exclusive to plans of Partition1. In essence, CE1 cannot be any subset of the events denoted by "X...leaf". If it were, CE1 would not be exclusive to the plans of Partition1. The only other possibility is that CE1 is empty, which contradicts our condition that every partition have a non-empty common-exclusive set. Therefore, this stuck situation cannot occur given valid and non-empty common-exclusive sets for every partition.

The above argument can also be used to show that we can relax the condition to (n – 1) partitions having a valid and non-empty common-exclusive set. That is, with n partitions, we can allow one partition to have an empty set and still not get into a stuck situation, providing that all other partitions have valid non-empty sets. In figure 4.5, we can assume that CE2 may be empty. But in this case no other common exclusive set can be empty. This still means that CE1 cannot be a subset of events from X to leaf, but it also cannot be empty, contradicting the relaxed condition. With this relaxed condition, eventually, clarification from Branchpoint X must still eliminate (n – 1) partitions before running out of events to ask. The situation in which we can get stuck is the case where two or more partitions would have empty common-exclusive sets. In such a case, it is possible to end up with plans in different partitions that are identical plans except for their non-clarified intervening events.

Another viewpoint of this safety determination is that we are attempting to determine if the faults of the remaining plans are solely based upon the different paths and actions that occur after the bottom.

4.7.2. Examples of Branch Point Skipping

As a simple example of branch point skipping, we again use the cooking example of 4.1, with the event hierarchy of figure 3.7. The user's question involves French wine and marinara sauce. If we list all events for each recognized plan, the result would be the following:

p1: (end, entertain guest, choose wine, red wine, French wine, prepare meal, pasta dish, fettucini marinara, marinara sauce).
p2: (end, entertain guest, choose wine, red wine, French wine, prepare meal, pasta dish, spaghetti marinara, marinara sauce).
p3: (end, entertain guest, choose wine, red wine, French wine, prepare meal, meat dish, chicken marinara, marinara sauce).
p4: (end, entertain guest, choose wine, white wine, French wine, prepare meal, pasta dish, fettucini marinara, marinara sauce).
p5: (end, entertain guest, choose wine, white wine, French wine, prepare meal, pasta dish, spaghetti marinara, marinara sauce).
p6: (end, entertain guest, choose wine, white wine, French wine, prepare meal, meat dish, chicken marinara, marinara sauce).

There are two fault partitions (n = 2). The first partition contains the faultless plans (p1, p2, p4, and p5). The second partition contains the two meat dish plans (p3 and p6) that have a fault concerning our vegetarian guest. In clarification, the event of *entertain guest* has two sub-branches. Therefore, we are at the point at which the sub-branch of *choose wine* is the CB and the sub-branch of *prepare meal* is on top of the branchpoint stack. The current branch point of *choose wine* has two alternatives of *choose*

red wine and *choose white wine*. The 'yes' result of each alternative is n partitions. That is, a 'Yes' reply to either red wine or white wine would leave us with two fault partitions (the same is true for a 'No' reply). This indicates that we may be able to skip branch points, so the BRANCH_POINT_SKIP procedure is utilized to determine the branch point to skip to.

The first step of the BRANCH_POINT_SKIP procedure is that of finding the diamond bottom. In tracing any path from CB we come upon the leaf event of *choose French wine*. Although all plans do converge on this event, it is a leaf event, and indicates that we are in the second case, for which the bottom is the top of the BPS. This is not a diamond configuration (the first case), because the event of convergence has no children. Since the branchpoint stack is not empty, the top of the stack is returned as the bottom (i.e., the event of *prepare meal* is the branch point to skip to).

The second step of the BRANCH_POINT_SKIP procedure determines the set of intervening disjunctive events between CB and the bottom. In this case, these events are *choose red wine*, *choose white wine*, and *choose French wine*.

The final step (SAFE_SKIP) is that of determining the safety of skipping branch points by establishing a common-exclusive set for each partition. Those disjunctive events common to all plans of the first partition are *entertain guest, French wine, and pasta dish*. The event of *entertain guest* has already been established and the event of *French wine* is an intervening event. Only the event of *pasta dish* occurs "after" *prepare meal*. Thus, the common-exclusive set for the first partition contains the single event of *pasta dish*. The set is exclusive to the plans of the first partition. Plans in the second partition (p3 and p6) do not contain the *pasta dish* event. Therefore, the first partition has a valid and non-empty common-exclusive set. The common-exclusive set of the second partition contains the two events of *meat dish* and *chicken marinara*. Both events are disjunctive events occurring after the bottom and are common to all plans of the second partition. Since the set is exclusive to the plans of the second partition, the set is valid. Since all partitions have a valid and non-empty common-exclusive set, we can safely skip branch points. The BRANCH_POINT_SKIP procedure pops the branch point stack and returns the result of *prepare meal* as the branch point to skip to. The next clarification question is then about the type of meal rather than the choice of wine. A 'Yes' or 'No' reply to either event of *pasta dish* or *meat dish*, will leave one remaining partition.

A case for not skipping can be shown with the same example, but different fault partitions. Assume three fault partitions as follows:

Partition1 has the plan with red wine and spaghetti marinara and the plan with white wine and fettucini marinara.

Partition2 has the plan with white wine and spaghetti marinara and the plan with red wine and fettucini marinara.

Partition3 has the two plans with chicken marinara.

If we had skipped branch points from *choose wine* to *prepare meal*, a 'Yes' reply to *pasta dish* and a 'Yes' reply to *spaghetti marinara* would leave us with the first two fault partitions each containing a single plan involving spaghetti marinara (but different wines), but no more questions to further clarify these plans (since we cannot backtrack to the choice of wine). However, in this case, our SAFE_SKIP procedure would have returned a 'false' result. The first two partitions do not have a valid and non-empty common-exclusive set. Although the event of *pasta dish* is common to all plans of the

first partition, it is not exclusive to the plans of that partition, since the plans of the second partition also include this event (and therefore, the set). In this case, our BRANCH_POINT_SKIP procedure would have returned a nil result, indicating that the choose wine branch point cannot be skipped and the type of wine should be clarified.

To illustrate more complex branch point skipping, figure 4.6 shows an event hierarchy with events simply labeled by letters and numbers. This event hierarchy represents a total of 96 possible plans. If we were to list the disjunctive events of the first and last plans, these would be (adfmov13) and (acknrw6). In this case, assume that the user's plan involves the disjunctive events: (aejnqv23). These events are circled in the figure.

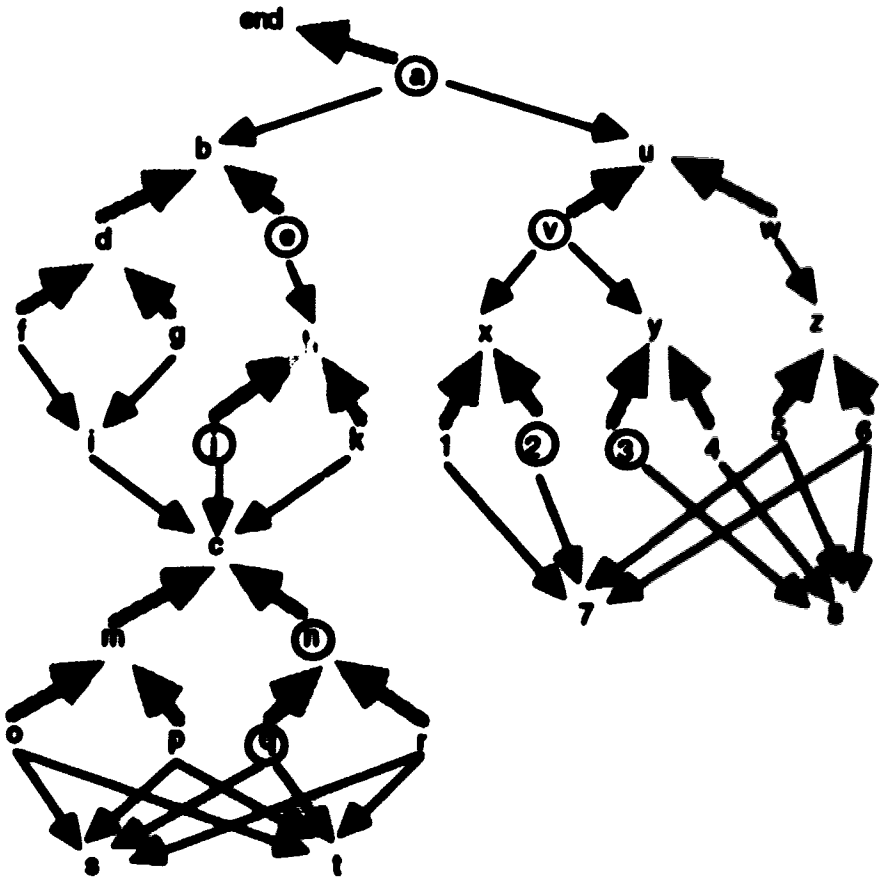

Figure 4.6 Event hierarchy to illustrate branch point skipping.

Figure 4.7 shows the fault partitions for the event hierarchy of figure 4.6. For each partition, we denote those events that are members of each plan of the partition. For example, the first partition contains all those plans having events *m* and *o*. All other disjunctive events of these plans can vary. As a result, the first partition will contain 24 plans. For the last partition, we list the disjunctive events of the plans in that partition.

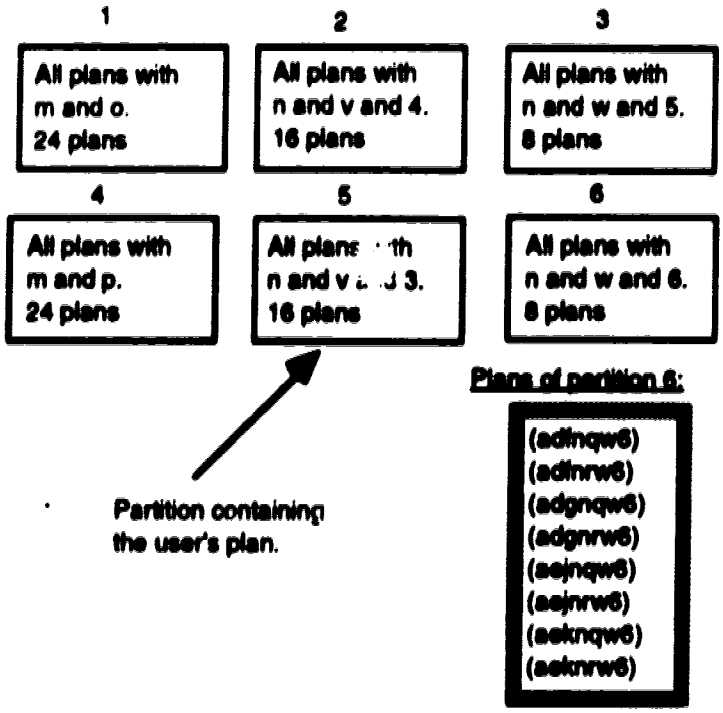

Figure 4.7 The fault partitions of the hierarchy of figure 4.6.

Without branch point skipping, a menu clarification of the event hierarchy of figure 4.6 would give us the following dialogue:

System: Do you intend to
d
e
User reply: e.

System: Do you intend to
j
k
User reply: j.
(At this point, we have 24 remaining plans, but we still have 6 fault partitions. Every partition contains plans which involve the events of e and j.)

System: Do you intend to
m
n
User reply: n.
(The reply results in the elimination of all plans with event m. This will eliminate two partitions (numbers 1 and 4) leaving 4 remaining partitions, all of which contain plans involving events e, j, and n.)

System: Do you intend to
q
r
User reply: q.
(At this point there are 6 remaining plans in 4 fault partitions:

Partition 2 has plans (aejnqv14) and (aejnqv24)
Partition 3 has the plan (aejnqw5)
Partition 5 has plans (aejnqv13) and (aejnqv23)
Partition 6 has the plan (aejnqw6)
The next branch point is event u, on the branch point stack.)

System: Do you intend to
v
w
User reply: v.
(There are now 4 plans in two partitions.)

System: Do you intend to
1
2
User reply: 2.
(Only 2 plans remain, but they are in different partitions)

System: Do you intend to
3
4
User reply: 3.
(Only one plan and one partition remain. Ambiguity no longer matters.)

The above dialogue required 7 menu-type clarification questions. Now, we consider the same situation utilizing branch point skipping:

Step 1: (CB = b and BPS contains event u)
Initially, clarification will begin with the current branch point of event *b*, and the branch point stack containing event *u*. The 'yes' result of each alternative of the current branch point is m, or 6 partitions. That is, events *d* and *e* are in every partition. This indicates that we may be able to skip the current branch point. In searching for a diamond bottom we trace one path downward in the hierarchy (e.g., path d-f-i-c), and find that all 96 plans converge on event *c*. This event is the diamond bottom. The intervening paths, between event *b* and event *c*, are all simple paths. The set of common-exclusive events for each partition consists of the events listed for each partition in figure 4.7. Each set is valid and non-empty. Since this is true, the BRANCH_POINT_SKIP procedure returns the diamond bottom of event *c* as the branch point to skip to. The current branch point is set to this event.

Step 2: (CB = c and BPS contains event u)
The current branch point has two alternatives of events *m* and *n*. The 'yes' result for event *m* is 2 partitions, while the 'yes' result of event *n* is 4 partitions. Since all 'yes' results do not equal 6 partitions, this branch point cannot be skipped. This will give us the first clarification question:

System: Do you intend to
m
n
User reply: n.

All remaining plans involve event *n* in 4 remaining partitions. Partition numbers 1 and 4 have been eliminated. The current branch point is set to event *n*.

Step 3: (CB = n and BPS contains event u)

The current branch point has two alternatives of *q* and *r*. In this case, the 'yes' result of both alternatives is n partitions (i.e., 4 partitions). It is possible that the current branch point may be skipped. In searching for a bottom, a single path from the current branch point (e.g., q-s) results in a leaf event with no children. Since the branch point stack is not empty, event *u* is returned as the bottom. The common exclusive sets for the remaining four partitions are the same, except that event *n* is no longer a member of any set. The common-exclusive sets are all valid and non-empty. The top of the branch point stack is removed and returned as the branch point to skip to. The result is that the current branch point is set to event u, and the branch point stack is now empty.

Step 4: (CB = u and BPS is empty)

The alternatives of the current branch point (events *v* and *w*) each have a 'yes' result of 2 partitions. Since n is 4 partitions, the current branch point cannot be skipped. Therefore, the second clarification question is asked:

System: Do you intend to
v
w
User reply: v.

The reply eliminates all plans involving event *w*, and two partitions. Although only two partitions remain (n = 2), there are 32 remaining possible plans. The remaining plans and partitions are shown in figure 4.8. Each plan is shown as a list of the disjunctive events that make up the plan.

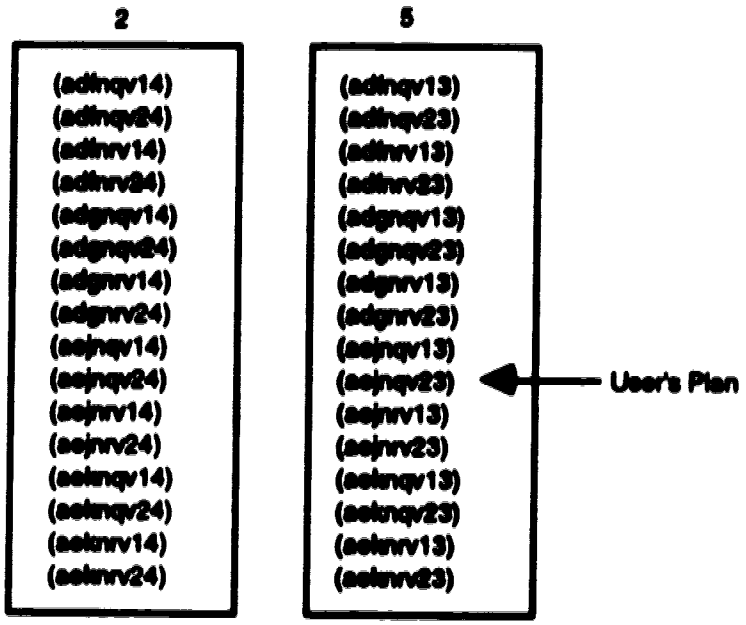

Figure 4.8 Remaining partitions for plans with event v.

The current branch point is set to the user's reply of event *v*.

Step 5: (CB = v and BPS is empty)

Since the current branch point has two conjunctive children, sub-branch *y* is placed on the branch point stack, followed by sub-branch *x*. Since there are no alternatives, sub-branch *x* becomes the current branch point.

Step 6: (CB = x and BPS contains event y)

The current branch point has two alternatives (events *1* and *2*), but the 'yes' result for each is n partitions. Both alternatives are in each remaining partition. In searching for a diamond bottom, we hit the leaf event of event 7. The branch point stack is not empty, so event *y* is returned as the bottom. The common-exclusive set for partition number 2 contains event *4*, while the common-exclusive set for partition number 5 contains event *3*. Both sets are valid and non-empty. Since this is true, the result of popping the BPS (event *y*) is returned as the branch point to skip to. The current branch point is set to event *y*, and the BPS is empty.

Step 7: (CB = y and BPS is empty)

The current branch point has two alternatives (events *3* and *4*). Since the 'yes' result of each is 1 partition, the third and final clarification question is asked:

System: Do you intend to
3
4
User reply: 3.

The current branch point is set to event *3*, but at this point ambiguity no longer matters. All plans with event *4* have been eliminated, along with partition number 2. There is only one remaining partition containing 16 plans, all of which have the same fault annotation. In this case, by skipping branch points, we only asked three menu-type clarification questions, instead of seven.

Before leaving the issue of branch point skipping, it should be noted that this possibility exists after 'No' replies to 'Yes/No' clarification questions; that is, before some alternative of CB becomes the next current branch point. For instance, the 'yes' result of all alternatives may be n partitions, but the branch point is not deemed to be "safe" for skipping. After a 'No' reply to the first alternative of the current branch point, it may be the case that it is safe to skip to another branch point. That is, there may be the possibility of skipping for the remaining alternatives of the CB. If we wish to pursue the possibility, we would check for branch point skipping after each 'No' reply to an alternative of the current branchpoint. Whether this may, or may not, be worthwhile is arguable, and depends on the event hierarchy configurations that may result from any given plan library. The same can be said for branch point skipping in general. We would argue that any system allowing multiple observations, with numerous complex plans, should also consider branch point skipping, to reduce interactive clarification dialogue; especially in those cases in which the number of possible plans far exceed the number of possible fault annotations.

4.8. Using Key Events

A proper definition of a key event is the following:

> A *key event* is the highest disjunctive event (in top-down order) that is a member of all of the plans that belong to a single particular fault partition, and is not a member of any remaining plans in any other fault partitions.

A fault partition may have several events that are shared by all plans of that partition, and no other plans. Among these events, the key event is the first such event in hierarchical top-down order.

When we ask the user about a key event, a 'Yes' reply will mean that the remaining plans are only those plans that have that key event. Since such plans are all located in one partition, ambiguity will no longer matter, and we can respond to the faults pertaining to that one partition. In other words, a 'Yes' reply to any key event will mean that clarification is over and we are done. On the other hand, a 'No' reply to a key event will mean that we eliminate all plans having that event. Since all such plans are in one, and only one, partition, we eliminate that partition. Should only one partition remain, ambiguity will not matter and we are done. However, if more than one partition remains, we have a problem. We must either return to the current branchpoint and select an alternative to ask, or we continue asking key events, if we have them.

The argument against re-clarifying from the current branchpoint is that the alternatives include all remaining plans, including those containing the key events. Either we ask some abstracted event of the key event, or we may switch context and coherence, in a confusing manner to the user. As an example, if our current branchpoint was preparing a meal, we might ask a key event of making spaghetti marinara. If the reply is 'No', and we return to our current branchpoint, the next clarifying question might be 'Are you making a pasta dish?', or it might be 'Are you making a meat dish?'. In the first case, the user may be wondering why we did not ask this question in the first place, whether or not a pasta dish is being considered. In the second case, the user may actually intend to make spaghetti pesto (instead of spaghetti marinara), and is left wondering why we would switch to asking about a meat dish. Also, should the user reply with a 'No' to making a meat dish (after a 'No' reply to making spaghetti marinara) the next clarification questions may be about several different types of pasta dishes. Switching back and forth between key events and the current branchpoint can easily lose the coherency of the direction of clarification questions. For this reason, we adopt the position that if we decide to ask a key event, we keep asking key events until only one partition remains, and we can respond. This means that if we have n fault partitions, we must have at least $(n - 1)$ key events.

If only two partitions remain, and at least one has a key event, then a 'Yes' or 'No' reply to that key event will leave only one partition remaining, and we are done. Should there be more than two partitions, having at least $(n - 1)$ key events, then we again have problems of coherence and context in the line of questioning. We could be "hopping" all over the event hierarchy. We can accept this to some extent, but we do not want to hop back and forth among different groups of plans. To exaggerate, we do not want to ask about a certain type of fruit such as apples, then ask about meat-eating dinosaurs, and then ask about oranges. Also, we do not want to ask about apples, then ask about fruit, and then again ask about oranges. In the first case we hop about the breadth of the hierarchy, and in the second, we hop about the depth of a particular branch of the hierarchy. One method of preventing this would be to group the key events according to the various alternatives of the current branchpoint. That is, all key events descending from a certain alternative of the current branchpoint are grouped together, and we ask all the key events from one group before asking the key events of another group. However, we might be better off asking the alternatives of the current branchpoint in the first place. A 'No' reply to a key event can only eliminate one partition. Asking about an alternative of the current branch point may eliminate more than one partition, in which case the hierarchical approach is better than asking a key event, and we do not enter into context and coherency problems. Even if we determine that the alternatives of the current branchpoint will not reduce any partitions, should we abandon the hierarchical approach

and use key events if we have enough of them? Perhaps a certain alternative will lead to a new current branchpoint with alternatives that greatly affect the number of partitions, or to another branchpoint that does not. This will depend on the user's plan. Once we choose to go to key events, we are forced into eliminating one partition at a time until we ask the correct key event. With any large number of fault partitions, we may be lengthening the number of clarification questions, instead of reducing them. There is also the question of it being worthwhile to check on all the possibilities of the average number of hierarchical questions versus the average number of key event questions. These problems with key events are as yet unresolved, so the method we currently propose restricts the use of key events to either two or three remaining partitions.

We only consider using key events when there are two or three remaining fault partitions, and we have key events for all but one of them. At most, we ask the user two more clarification questions. At most, we make one "hop" and avoid the possibility of going back and forth among different groups of plans in our line of questioning. We check for the possibility of using key events before clarification begins. We also check after every clarification question, providing that ambiguity still matters.

The key event procedures (for 3 fault partitions) are:

```
procedure KEY_EVENTS_POSSIBLE(branchpoint)
begin
        if (n > 3) then  return (false) /* n is the partition number */
        else key_events ← FIND_KEY_EVENTS(branchpoint)

        if (number of key_events < (n – 1)) then return (false)
        else return (true)
end
```

The procedure of FIND_KEY_EVENTS returns a set of *key event-partition* combinations so that each key event is associated with its particular partition. One possible method for finding a key event of a partition is that of selecting a single plan of the partition and tracing its path (from the CB) through the hierarchy. At each disjunctive event of the trace, we check the plans (plan labels) of that event. If the event is shared by all plans of the particular partition, and not held by any other remaining plans, then the first such event is a key event for the partition. A similar strategy may be employed in finding the set of common-exclusive events for branch point skipping.

```
procedure KEY_EVENTS
begin
        while (n > 1) do:
          begin
            event ← an event from key_events
            Ask user about event
            if reply is 'Yes' then
                remove all plans not having event
                remove all partitions except the one with event
                n ← 1
            else
                remove all plans with event
                remove partition with event
                n ← (n – 1)
          end
        return
end
```

To further illustrate the use of key events, we can use the hierarchy of figure 3.3 with the fault partitions shown in figure 4.9.

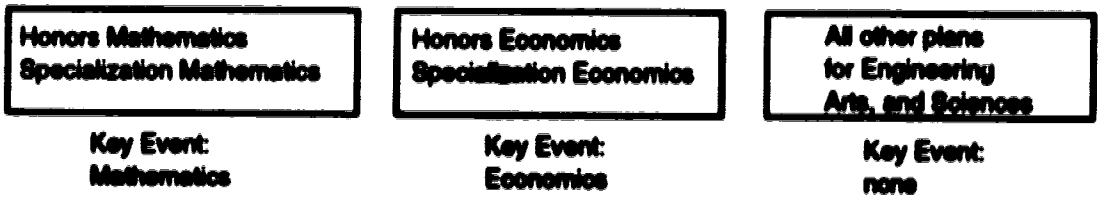

Figure 4.9 Key events with three remaining fault partitions.

Given the partitioning of figure 4.9, it is possible to use key events immediately, since there are three partitions and two have key events. In traversing the hierarchy of disjunctive events for a plan of the first partition, the events of *BSc. degree* and *Mathematical Sciences* are not key events (the third partition contains Computing Science and Statistics plans that also abstract to *Mathematical Sciences*). The event of *Mathematics* is a key event of the first partition, since all plans of the partition contain the event, but no other plans do. Similarly, the event of *Economics* is a key event for the second partition. Since it is possible to use key events, the first clarification question would be about entering the department of Mathematics. If the reply were to be 'No', then the second and final question would be about entering the department of Economics. A 'Yes' or 'No' reply to this last question will leave one remaining partition. Is this better than the hierarchical approach? It depends on the plan of the user. The advantage of key events is that there is a guarantee of only asking a certain number of clarification questions.

Given two fault partitions and a key event, we can do no better than asking one more clarification question. Given three fault partitions and at least two key events, we are guaranteed a maximum of two more clarification questions. By following the

hierarchical approach, we may only need one clarification question. For example, with the fault partitioning of figure 4.9, and a user's plan to obtain an Engineering degree, a single menu question about the degrees, would have left one remaining partition. However, with any hierarchy involving "depth", and a user's plan that is closely related to a key event, we could clarify branch point after branch point, (reducing plans, but not partitions) until we get to a branch point that has a key event as an alternative. That is, we can do much worse than two more clarification questions. A decision to use key events with three remaining partitions, rather than two, may largely depend upon the depth of the plan library and whether or not menu-type questions are used.

4.9. Shared Events

A key event is a disjunctive event of all plans of one fault partition and is not a member of any other possible plans in any other fault partition. A step beyond this is having a disjunctive event common to all plans of more than one partition (but not all partitions), and not held by any other remaining plans. We term such events to be *shared events*. Asking about a shared event will eliminate or substantiate more than one partition at a time. For example, if we had six fault partitions and a shared event for three of those partitions, then a 'Yes' or 'No' reply to that event would leave us with three remaining partitions. We can imagine an ideal situation with n partitions in which we always have a shared event for (n/2) partitions. In such a case, 2^x partitions would require x clarification questions. Can we make a case for using shared events?

The first problem to tackle is that of finding shared events, but this would be similar to (and may perhaps be combined with) whatever method is used to find key events. We may find more than one shared event for any particular set of partitions. In this case we would like the shared event to be the highest such event in order to be consistent with our top-down approach. Assuming that we find more than one shared event (where each covers a different set of partitions), then we might consider ranking them according to how closely they approach (n/2) partitions.

Another problem is that of knowing exactly when to use shared events. If an alternative of the current branchpoint eliminates more than one partition on a 'Yes' or 'No' reply, then that alternative is a shared event, but we do not consider it as such in our discussion of shared events. That is, if we talk about switching to, or using shared events, we imply leaving the hierarchical approach and the current branchpoint and asking about some shared event lower in the hierarchy (although this appears to be confusing, the following sentence now makes sense). We should not leave the hierarchical approach and ask about a shared event if an alternative of the current branchpoint does just as well. There is also the aspect of better prospects with the next branchpoint. That is, we don't want to go to shared events if the entire hierarchical approach gets us just as far with the same number of clarification questions (but, we have to somehow determine this, or take a chance).

Assuming we leave the hierarchical approach and ask about a shared event, we have further complications. The reply will either be 'Yes' or 'No'.

In the case of a 'No' reply, we remove the plans and partitions of the shared event. If ambiguity still matters and we have more than one remaining fault partition, then we are in one of the following two cases:

1. We have more shared events or key events that we can ask.
2. We do not have any shared events or key events for the remaining partitions.

We want to avoid the second case because it means returning to the current branchpoint and clarifying from there. In order to avoid this, we need to check out the remaining partitions for shared and key events for a 'No' reply, making sure that we have enough to take us all the way down to one remaining partition. Also, shared events will have similar coherency and context problems as with using key events.

A 'Yes' reply to a key event will mean that ambiguity no longer matters. However, a 'Yes' reply to a question about a shared event will leave us with more than one remaining partition. If we do not have further shared events or key events for these remaining partitions, then we must go back to our hierarchical approach. In this case we set the current branchpoint to be the shared event we just asked the user. The problem here, is that we may completely bypass necessary conjunctive branch points. Figure 4.10 illustrates this.

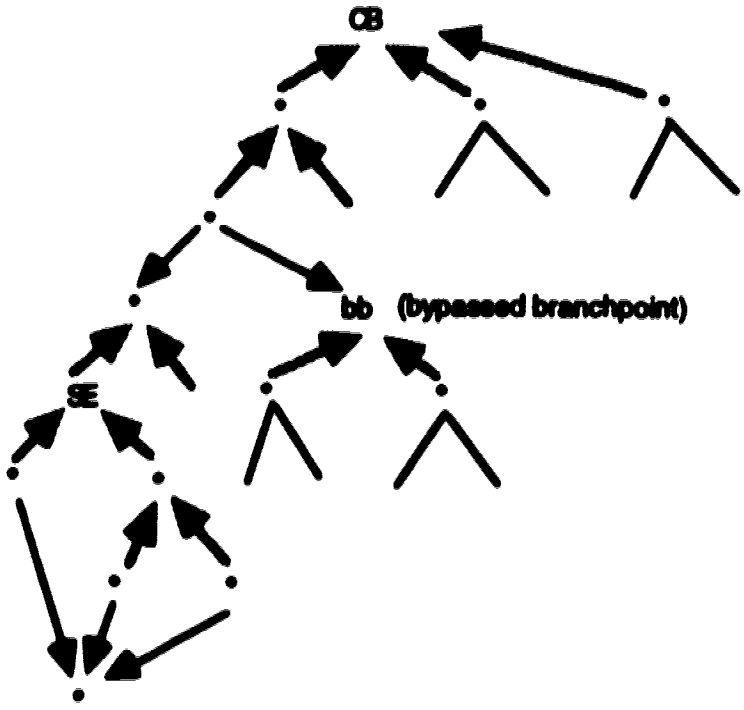

Figure 4.10 Bypassing a branchpoint with shared events.

In figure 4.10, assume we get a 'Yes' reply to the shared event of *SE*. This event is shared by plans in different fault partitions. Assuming we did not have any more key events to ask, we would set the current branchpoint to be the event *SE* and resume clarification from there. The problem with this is that we do not place event *bb* on the branchpoint stack. This means that we must track the events on the path from CB to *SE*, placing such missed conjunctive events on the branchpoint stack.

In summary, the problems with using shared events are:
- maintaining coherency and context.
- making sure we have enough shared events and key events to finish the job, or making sure our branchpoint stack is maintained correctly.
- establishing when to use them, that is, when is the hierarchical approach not as good.
- establishing how often we should check for shared events and will it be worthwhile to do so.

Many of these problems are similar to those for key events. They are mentioned here, but not yet resolved to the point that we may confidently propose a proper and consistent method for their use, other than our limited use of key events. Certainly the advantages of using such shared events in combination with key events has the potential for greatly reducing the number of clarification questions asked, but their proper exploitation is as yet unresolved and left to future research.

4.10. Overriding Faults

It may be the case that certain faults have much more impact than others. If a fault is associated with a certain disjunctive event (e.g., because of a constraint violation of that event), and that fault is deemed sufficient to advise the user against pursuing that event, then it is also sufficient to advise the user against pursuing any specialization of that event (and specializations of that specialization and so on). The user should not be following a certain course of action, no matter what the particular plan may be.

To quickly review the example of the previous chapter, assume that we are preparing a meal to entertain a guest. The guest is a vegetarian and the system knows this, and this fact violates a constraint for the event of *make meat dish*. There may be several possible types of meat dishes, but the exact type does not matter if a meat dish should not be made in the first place.

Such faults, which make an event an inappropriate course of action, we term to be *overriding* faults. The fault has such importance as to override the faults of diverging paths below the event. We should not be clarifying the specializations of such an event (even though their ambiguity matters and we still have more than one partition), when the user should not be doing the event in the first place. For example, if we know the user is intending to make a meat dish for a vegetarian guest, and chicken marinara is faultless, but shrimp marinara is not, should we ask 'Are you intending to make shrimp marinara?'. We may ask about a lot of meat dishes before ambiguity no longer matters, but our response to any meat dish will contain a comment about the guest being a vegetarian, which leaves the user wondering why we kept on asking about all those different meat dishes.

If we know that the user is intending to pursue an event that has an overriding fault, then we should halt further clarification dialogue and respond with a response that addresses the overriding fault. We do this even though ambiguity of the remaining plans still matters, in the sense that there may still be more than one fault partition remaining.

We can easily incorporate this concept in our clarification process. Overriding faults can be associated with the events that produce them during the critiquing process. Whenever we check to see if ambiguity still matters, we also check the event that is to be the next current branchpoint to see if it has an overriding fault. If the event has an overriding fault, then we can terminate clarification and provide a response to address the overriding fault.

As an example, we can use the event hierarchy of figure 3.3 with an observation of a course, CMPUT 199, on advanced programming languages. The course involves intensive programming concepts for non-procedural programming languages such as Lisp, Prolog, and object-oriented languages. Figure 4.11 shows the fault partitioning, with each partition having fault labels. The fault labels with capital letters and a numerical designation of 0 represent overriding faults.

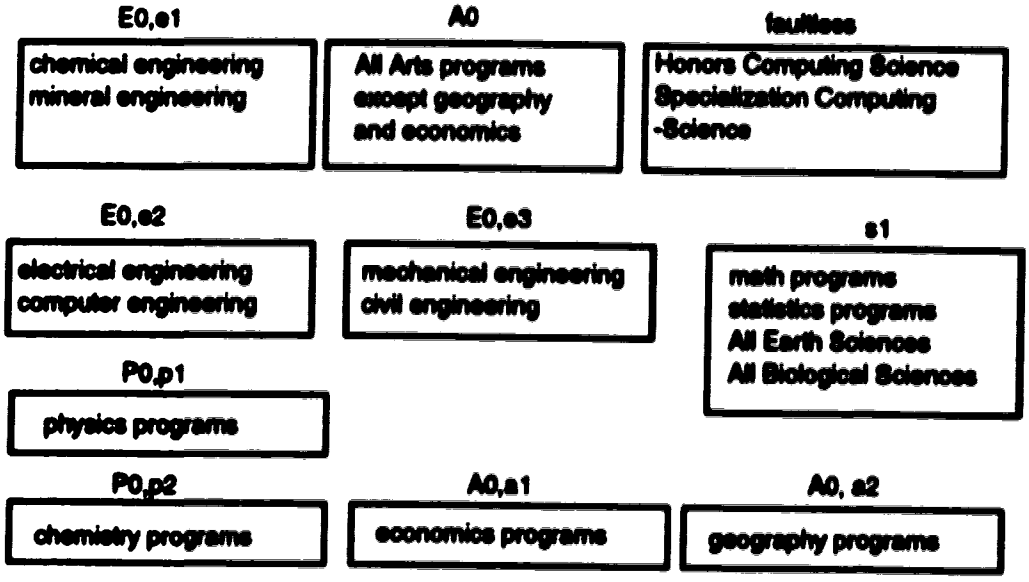

Figure 4.11 Overriding faults.

For clarity, assume the following brief and partial responses for each fault label of figure 4.11:

E0: All first year Engineering students are required to take a FORTRAN programming course. (The fault results from a failed precondition for the event of *Engineering degree*. All computer courses must be courses involving FORTRAN programming. It is an overriding fault, in that all Engineering students should not be taking this particular course.)

e1: Chemical and Mineral Engineering fields require the computer engineering course ENCMP 102 on advanced FORTRAN.

e2: Electrical and Computer Engineering fields require the computing science course CMPUT 185 on basic FORTRAN.

e3: Civil and Mechanical Engineering fields require the computer engineering course ENCMP 101 on basic FORTRAN.

A0: This particular course involves intensive programming and is generally not recommended for first year Arts students. See your particular department advisor if you wish to take the course. (The response results from a failed precondition for the event of *BA. degree*. Only introductory or application computer courses are advised for first-year computer requirements and computer electives. This is an overriding fault.)

a1: Economics students must take CMPUT 171, on spread-sheet applications.

a2: Students in Geography must take the course CMPUT 181 on graphics and imaging applications.

P0: Students in the Physical Sciences must take a procedural-language computer programming course. (A course in advanced programming languages violates a precondition for the event of *Physical Sciences*, which in our case

involves the two departments of Chemistry and Physics. This is an overriding fault.)

p1: Physics students are strongly recommended to take computer course CMPUT 186 on basic Pascal programming.

p2: Chemistry students are strongly recommended to take computer course CMPUT 187 on basic C programming.

s1: The Science department of [*variable*], recommends the introductory course CMPUT 162.

Given the above fault partitioning (40 possible plans divided among 10 possible partitions), our clarification process would terminate (even though more than one partition would remain) should we establish that the user intends to obtain an Engineering degree, or a Bachelor of Arts degree, or intends to enter the Physical Sciences. In each of these cases, an overriding fault is associated with the event. The specializations of the event are not explored even though their ambiguity still matters. For example, if our first question establishes that the user intends to obtain an Engineering degree (leaving three remaining partitions), we do not further clarify which engineering field is being pursued, but provide the response for the fault label of E0. The result would be similar, had the user's plan involved a Bachelor of Arts degree, but in this case the response would have been that for the fault of A0, without further clarification of the area of Arts studies (such as the humanities, social sciences, or fine arts) or the particular department (such as the social sciences of Economics, Geography, or History).

In effect, we can view the above Engineering plans, in the different partitions, to be collected into one partition with the single fault label of E0. This view can be made more concrete by having the critiquing process consider overriding faults as being the only fault of a plan. In this case, the partitions and fault labels of figure 4.11 would be reduced to those shown in figure 4.12. This would give the clarification process fewer partitions to deal with.

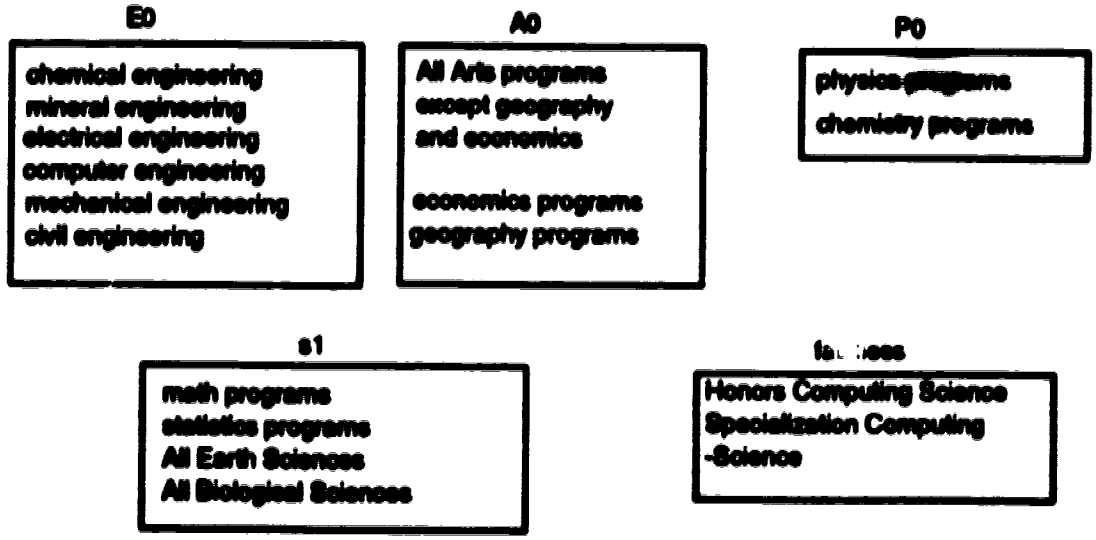

Figure 4.12 Collecting Plans with overriding faults

4.11. Complex Clarification Questions and Responses

The ground rules we have set only allow us to ask 'Yes/No' questions about single events and to respond when we have only one remaining partition (except in the case of overriding faults). The ground rules establish a basic starting framework for dealing with clarification dialogue. Here, we look at possibilities of relaxing these ground rules.

If we ask the user a 'Yes/No' question about more than one event, we must somehow combine the events using 'and' and 'or' connectives. We can ask about a group of plans by combining events unique to those plans. For example, if we have a partition containing two out of three possible pasta plans, we might ask 'Are you making spaghetti marinara or spaghetti pesto?'. We may ask about a single complex plan that has no unique event by combining events which distinguish that plan from any other. For example, if we have one plan in one partition, and that plan has no unique event (or key event), we might ask 'Are you serving red wine and making spaghetti marinara?'. It can be seen that such questions may reduce the total number of clarification questions required, when there are only a few remaining partitions. However, it can be argued that such questions can be quite confusing for the user, and this aspect gets worse as we start combining more events into a single question. For example, if we have two wines and two pasta dishes and the user is considering a plan to serve red wine and make spaghetti pesto, consider the following two dialogues with respect to clarity:

Dialogue 1 (Questions about single events):

Do you intend to serve red wine? Yes.
Are you making spaghetti marinara? No.
(the user is serving red wine with spaghetti pesto).

Dialogue 2 (Questions about more than one event):

Do you intend to serve red wine and make spaghetti marinara or do you intend to serve white wine and make spaghetti pesto? No.
Do you intend to serve red wine and make spaghetti pesto? Yes.
(as opposed to white wine and spaghetti marinara).

In general, as we increase the events combined into a single 'Yes/No' question, we increase the length of the question, its "confusion factor", and the user's response time. These factors should be balanced with reducing the number of clarification questions. There is a trade-off between the number of questions and the ease of understanding of the questions. Exact rules for when and how to ask such complex questions need to be examined, empirically tested and resolved.

It is possible to respond to the faults of more than one remaining partition. That is, we could end clarification at a certain point and respond to more than one partition. For example, if we had only two remaining partitions and each contained a single plan, our response would be something like: 'If you are intending to do plan *A*, then *fault of plan A*, but if you are doing plan *B*, then *fault of plan B*'. Here again, we need to establish certain rules if we wish to avoid overly verbose responses. The user is interested in the faults that apply to the partition containing the user's plan, and is probably not interested in learning or reading about all the faults of other plans. We must consider several aspects with respect to the length and verbosity of the response:

- We have to consider the number of fault partitions that the response is intended to cover.
- We need to consider the number of faults for any partition.
- We must consider the number of different plans in each partition, since each must be described so users know which faults apply to their particular plan.
- We need to consider the number of events required to adequately describe and distinguish each plan.
- Lastly, we need to consider when a lengthy response is better than asking one or two more clarification questions.

Lastly, there is the possibility of using non-homogenous menus. But here again, certain aspects must be considered about the length and clarity of such menus. In the case of complex plans, for which individual events do not differentiate one plan from another, each menu item may have to describe more than one event. The same is true for plan libraries in which the relationship between events may not be clear to the user. A series of events may be required to adequately describe a single plan to the user. A lengthy menu of such descriptions may not be as desirable as several shorter and clearer menus.

Although we can see certain advantages with the use of complex clarification questions and complex responses, the cases are arguable. We still require further research into this area.

4.12. Application Specific Clarification

In this chapter we have described a general clarification process, in which several different concepts have been unified in order to show their integration. We do not intend to imply that the process should be taken as a whole and applied to every situation. In the interest of increasing the efficiency of clarification, we need to consider the usual event hierarchies that result from specific applications.

Specific applications may result in event hierarchies for which some aspects of the clarification process are more important than others. For example, applications that usually result in short and wide hierarchies with menu-type clarification questions, will find little benefit in using key events. On the other hand, applications that result in long thin hierarchies with several branch points, but generally few fault partitions, may find the use of key events (and the concept of shared events) to be extremely valuable. Applications with plan libraries that contain complex plans or naturally occurring diamond configurations, will benefit from a branch point skipping mechanism, but applications without these, will have little use for branch point skipping. In short, clarification should be tailored to specific applications.

Chapter 5. Summary and Future Research

5.1. Contributions of this Thesis

Advice-giving systems that use plan recognition, for the purpose of providing a cooperative response to users of the system, must deal with ambiguity of recognized plans, should more than one plan explain a set of observations. For interactive systems, one method of resolving the ambiguity is that of clarifying the plans with the user. That is, the system takes control, and asks the user certain questions about the possible plans, in order to determine the user's plan. A cooperative response can then be made in light of the user's plan. It is desirable to keep such clarification dialogues to a minimum. That is, as far as possible, the system's clarification questions should be concise, clear, and few in number.

In their work on plan ambiguity and cooperative response generation, van Beek and Cohen (see [van Beek and Cohen 91] and [van Beek *et. al.* 93]) propose solutions as to *when* the system should clarify, and *what* the system should ask. These solutions involve a critiquing component and a hierarchy of events to represent the recognized plans. The critiquing component assigns faults to the possible plans, which are partitioned according to these faults. The system enters into a clarification dialogue only when the faults of possible plans differ, and their ambiguity matters to the eventual response. Should the possible plans all have the same fault, a response can be given to address the fault without determining the exact plan of the user. Should the faults of the plans differ, the system clarifies by asking about events in the hierarchy representing the possible plans. In top-down fashion, the user is asked about the different alternative choices of each disjunctive branch point in the hierarchy, until all remaining plans have the same fault. The individual clarification questions are concise and clear, whether using 'Yes or No' type questions about a single alternative, or menu questions about the alternatives of a single branch point. This thesis has built upon this basic approach to clarification dialogue.

The particular contributions are the following:

1. We have provided a method of clarification that can deal with complex hierarchical plans. Such plans involve multiple necessary components (necessary sub-plans or sub-branches required to achieve the goal of the plan). By maintaining the event hierarchy in a tree-like structure, and placing conjunctive children onto a branch point stack, we can clarify the alternatives of different sub-branches of complex plans in a depth-first manner.

2. We have proposed a criterion as to *when* we should look for the possibility of skipping branch points in the event hierarchy. Whenever each alternative of a current branchpoint is present in all remaining fault partitions, there is the possibility of skipping that branch point. Should this not be the case, some alternative makes a difference to the remaining partitions (and the response), and the branch point should not be skipped.

3. Given the possibility of skipping branch points, we have provided a method of determining when we *can* skip branch points, in the case that plans converge to a single event and in the case they do not. In the former case, the branch point to skip to is the event at which the plans converge. In the latter case, the branch point to skip to is the next event on the branchpoint stack. In either case, we determine if the ambiguity can be resolved (so that it no longer matters) by only asking those events occurring after the branch point we would skip to. If so, the response does not depend upon clarifying any intervening event.

4. We have proposed a set of rules to determine the order in which the different alternatives of a current branch point are clarified, in the case that menu clarification questions are not used and the likelihood of plans cannot be determined. By first determining the remaining partitions for a 'Yes' and 'No' reply to each alternative, the user is asked that alternative leaving the fewest remaining partitions on average. The strategy attempts to eliminate the number of partitions more quickly than with a random selection.

5. We have described the concept (and limited use) of key events that "isolate" the plans of a particular fault partition. Asking the user about such events will either substantiate or eliminate all plans of a fault partition. We have also described some of the problems with their use. We have also briefly described an extension of this concept to that of shared events that isolate more than one partition, but as yet, the proper use of this concept has not been determined.

6. We have described a method that avoids pointless clarification in the case that the user is intending a certain action that is inappropriate for an observation. An overriding fault is associated with such an action, and a response is provided which addresses the overriding fault. The ambiguity of remaining plans may still matter to providing a cooperative response for the specializations of the inappropriate action, but these specializations are not further clarified.

7. We have combined and unified the above methods into one general clarification process, considering both menu-type clarification questions, as well as 'Yes/No' questions. We have also given general algorithms for the procedures involved in this process, as well as the data requirements and the changes that occur to this data.

5.2. Open Problems and Future Directions

The different aspects of plan recognition, critiquing, and clarification must be integrated and implemented for testing on a large number of plans. Empirical testing with a range of different users is required to adequately assess issues of efficiency and ease of use. The work presented here provides a general framework for clarification with general algorithms and a proposal for data requirements. The next step is that of implementation and integration within a complete system. Research is currently underway, at the University of Alberta, toward a complete system. The system combines a menu-based front-end for user questions, Kautz's plan recognition system, a critiquing component, and a clarification implementation based on our original trial program and the strategies developed here. Once complete, the system can be empirically tested on many users, and its efficiency can be accurately assessed with respect to the number of plans it can handle.

With respect to the clarification process, we believe that further exploration of the use of key and shared events may provide promising results. The idea behind this is simple in concept, in that we always ask the user about that event which will leave the fewest remaining partitions on either a 'Yes' or 'No' reply. The particular event does not have to be an alternative of the current branch point, but can be any disjunctive event in the given event hierarchy. Currently, there are two major obstacles to this approach. One is that of maintaining a coherent direction of clarification questioning. The second problem is that of determining if the hierarchical approach should be maintained. That is, fewer overall clarification questions may result from "sticking to" the hierarchical approach, even though the alternatives of the current branch point do not leave fewer partitions than some shared event lower in the hierarchy. Closely related to this, is the possibility of using a bottom-up approach and determining if this approach will reduce ambiguity faster than the top-down method.

By relaxing the ground rules for our clarification questions and the response, the flexibility and minimization of clarification dialogue can be enhanced, at the expense of clarity to the user. The degree of clarity must be considered. However, in certain situations, we can visualize much shorter clarification dialogues. For example, the following dialogue consists of a single non-homogenous menu question, in which each menu item isolates or covers one or more fault partitions (e.g., by shared events, key events, or combining the two), and the response covers two partitions.

User: Should I take course X?

System: Do you intend to
1. Obtain an Engineering degree or a BSc. in Physical Sciences?
2. Obtain a Bachelor of Arts degree?
3. Obtain a BSc. in the area of Earth or Biological Sciences?
4. Enter the department of Computing Science?
5. Take an Honors program in Mathematics or Statistics?
6. Take a Specialization program in Mathematics or Statistics?

User reply: 4.

System response: If you are taking the Honors program in Computing Science, the department prefers course Y over course X. However, course X is a good choice for the Specialization program.

Although clarification dialogue may be dramatically reduced, the issue of clarity is of prime importance. If we can ask complicated questions, then rules must be established to maintain clarity, and they must be empirically tested on a wide range of users.

As far as we can tell, the proper design of a plan library is still a "black art". The hierarchy of events that make up a plan library is meant to represent the possible actions and plans of a particular domain. However, there are no rules, methods, or tools available, that can aid in the design and configuration of the hierarchy. The designer must transform the domain into a hierarchy, that can encompass the actions of the domain, without the benefit of any established methods. From our own experience, in our course-advisor domain, we have found that how the hierarchy is organized can greatly influence the number of clarifying questions. However, we believe much more work can be done in this area of hierarchy design, to provide general methods and tools that can be applied to any domain.

Lastly, there are many issues that are application specific. A system using a graphical user interface might consider only using hierarchical pop-up menus to clarify a user's plan. Branch point skipping would be an important issue in limiting the set of menus presented to the user, and this may influence further work in hierarchy design for such systems. Systems having a natural language component must contend with issues related to combining events and faults into smooth and clear text, to avoid the "stilted" and awkward results that may occur from using "canned" clarification questions and responses. Increasing the efficiency of plan recognition, critiquing, and clarification for specific applications are dependent upon the scale, purpose, and type of application.

References

[Allen 83] Allen, J.; Recognizing intentions from natural language utterances; In Computational Models of Discourse, Michael Brady and Robert C. Berwick, (editors), pages 107-166, MIT Press, Cambridge, Mass., 1983.

[Allen and Perrault 80] Allen, J. and Perrault, C.; Analyzing intention in utterances; *Artificial Intelligence*, 15: 441-458, 1980.

[Buchanan and Shortliffe 84] Buchanan, B. and Shortliffe, E.; Rule-based expert systems. The MYCIN experiments of the Stanford heuristic programming project; Buchanan B. and Shortliffe, E. (editors), Addison-Wesley Publishing Company, Reading MA., 1984.

[Calistri-Yeh 91] Calistri-Yeh, R.; Utilizing user models to handle ambiguity and misconceptions in robust plan recognition; *User Modeling and User Adapted Interaction* 1(4): 289-322, 1991.

[Carberry 83] Carberry, S.; Tracking user goals in an information-seeking environment; *Proceedings of the Third National Conference on Artificial Intelligence*, pages 59-63, 1983.

[Fikes and Nilsson 71] Fikes, R. and Nilsson, N.; STRIPS: a new approach to the application of theorem proving to problem solving; *Artificial Intelligence* 2, 1971.

[Gaasterland 92] Gaasterland T.; *Generating cooperative answers in deductive databases*; PhD thesis, University of Maryland, 1992.

[Genesereth 79] Genesereth, M.; The role of plans in automated consultation; *Proceedings of the Sixth International Joint Conference on Artificial Intelligence*, pages 311-319, 1979.

[Goodman and Litman 92] Goodman, B. and Litman D.; On the interaction between plan recognition and intelligent interfaces; *User Modeling and User-Adapted Interaction* , 2(1-2): 83-115, 1992.

[Joshi, Webber and Weischedel 84] Joshi, A., Webber, B. and Weischedel, R.; Preventing false inferences; *Proceedings of COLING-84*, pages 134-138, Stanford CA., July, 1984.

[Joshi *et. al.* 84] Joshi, A., Webber, B. and Weischedel, R.; Living up to expectations: computing expert responses; *Proceedings of the Fourth National Conference on Artificial Intelligence*, pages 169-175, Austin, TX., August 1984.

[Kass and Finin 88] Kass, R. and Finin, T.; Modeling the user in natural language systems; *Computational Linguistics*, 14: 5-22, Sept. 1988.

[Kautz and Allen 86] Kautz, H. and Allen, J.; Generalized plan recognition, *Proceedings of the Fifth National Conference on Artificial Intelligence* , pages 32-37, Philadelphia, 1986.

[Kautz 87] Kautz, H.; *A Formal Theory of Plan Recognition* ; PhD thesis, University of Rochester, 1987. Available as: Department of Computer Science Technical Report 215.

[Pollack 86] Pollack, M.; A model of plan inference that distinguishes between the beliefs of actors and observers; *Proceedings of the 24th Conference of the Association for Computational Linguistics*, pages 207-214, New York, 1986.

[Sidner and Israel 81] Sidner, C. and Israel, D.; Recognizing intended meaning and speaker's plans; *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, pages 203-208, Aug. 1981.

[Sidner 83] Sidner, C.; What the speaker means: the recognition of speaker's plans in discourse; *International Journal of Computers and Mathematics*, 9: [illegible]

[Tennant 87] Tennant H.; Menu-based natural language: Encyclopedia of Artificial Intelligence, S. C. Shapiro (editor), John Wiley & Sons, pages 594-[illegible].

[van Beek 87] van Beek, P.; A model for generating better explanations; *Proceedings of the 25th Conference of the Association for Computational Linguistics*, pages 215-220, Stanford, Calif., 1987.

[van Beek and Cohen 91] van Beek, P. and Cohen R.; Resolving plan ambiguity for cooperative response generation; *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, pages 938-944, Sydney, Australia, 1991.

[van Beek *et. al.* 93] van Beek, P., Cohen, R., and Schmidt, K.; From plan critiquing to clarification dialogue for cooperative response generation; *Computational Intelligence*, 9: 132-154, May 1993.

[Webber 87] Webber, B.; Question answering; Encyclopedia of Artificial Intelligence, S. C. Shapiro (editor), John Wiley & Sons, pages 814-822, 1987.

[Wu 91] Wu, D.; Active acquisition of user models: implications for decision-theoretic dialog planning and plan recognition; *User Modeling and User-Adapted Interaction*, 1(2): 149-172, 1991.