

1

RADC-TR-88-11, Vol V (of eight)
Interim Technical Report
June 1988



AD-A199 824

NORTHEAST ARTIFICIAL INTELLIGENCE CONSORTIUM ANNUAL REPORT 1986

Building an Intelligent Assistant: The Acquisition, Integration, and Maintenance of Complex Distributed Tasks

Syracuse University

V. Lesser, W. B. Croft and B. Woolf, et. al.

DTIC
ELECTE
OCT 07 1988
S D
dyH

This effort was funded partially by the Laboratory Director's fund.

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss AFB, NY 13441-5700

88 10 6 014

This report has been reviewed by the RADC Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-88-11, Volume V (of eight) has been reviewed and is approved for publication.

APPROVED:



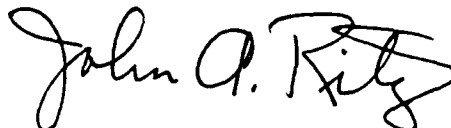
DOUGLAS A. WHITE
Project Engineer

APPROVED:



RAYMOND P. URIZ, JR.
Technical Director
Directorate of Command & Control

FOR THE COMMANDER:



JOHN A. RITZ
Directorate of Plans & Programs

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COES) Griffiss AFB NY 13441-5700. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notice on a specific document requires that it be returned.

NORTHEAST ARTIFICIAL INTELLIGENCE CONSORTIUM ANNUAL REPORT 1986, Building an Intelligent Assistant:
The Acquisition, Integration, and Maintenance of Complex Distributed Tasks, Vol V

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED		1b. RESTRICTIVE MARKINGS N/A	
2a. SECURITY CLASSIFICATION AUTHORITY N/A		3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.	
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A		4. PERFORMING ORGANIZATION REPORT NUMBER(S) N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) N/A		5. MONITORING ORGANIZATION REPORT NUMBER(S) RADC-TR-88-11, Vol V (of eight)	
6a. NAME OF PERFORMING ORGANIZATION Northeast Artificial Intelligence Consortium (NAIC)	6b. OFFICE SYMBOL (if applicable)	7a. NAME OF MONITORING ORGANIZATION Rome Air Development Center (COES)	
6c. ADDRESS (City, State, and ZIP Code) 409 Link Hall Syracuse University Syracuse NY 13244-1240		7b. ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700	
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Rome Air Development Center	8b. OFFICE SYMBOL (if applicable) COES	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F30602-85-C-0008	
8c. ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700		10. SOURCE OF FUNDING NUMBERS	
		PROGRAM ELEMENT NO. 62702F (over)	PROJECT NO. 5581
		TASK NO. 27	WORK UNIT ACCESSION NO. 13
11. TITLE (Include Security Classification) NORTHEAST ARTIFICIAL INTELLIGENCE CONSORTIUM ANNUAL REPORT 1986 Building an Intelligent Assistant: The Acquisition, Integration, and Maintenance of Complex Distributed Tasks			
12. PERSONAL AUTHOR(S) V. Lesser, W. B. Croft and B. Woolf, et al			
13a. TYPE OF REPORT Interim	13b. TIME COVERED FROM Jan 86 to Dec 86	14. DATE OF REPORT (Year, Month, Day) June 1988	15. PAGE COUNT 290
16. SUPPLEMENTARY NOTATION This effort was performed as a subcontract by the University of Massachusetts to Syracuse University, Office of Sponsored Programs. (over)			
17. COSATI CODES		18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)	
FIELD	GROUP	Artificial Intelligence Plan Recognition	
12	05	Intelligent Interfaces Intelligent Computer-Aided	
		Planning Instruction	
19. ABSTRACT (Continue on reverse if necessary and identify by block number) The Northeast Artificial Intelligence Consortium (NAIC) was created by the Air Force Systems Command, Rome Air Development Center, and the Office of Scientific Research. Its purpose is to conduct pertinent research in artificial intelligence and to perform activities ancillary to this research. This report describes progress that has been made in the second year of the existence of the NAIC on the technical research tasks undertaken at the member universities. The topics covered in general are: versatile expert system for equipment maintenance, distributed AI for communications system control, automatic photo interpretation, time-oriented problem solving, speech understanding systems, knowledge base maintenance, hardware architectures for very large systems, knowledge-based reasoning and planning, and a knowledge acquisition, assistance, and explanation system. The specific topic for this volume is the development of intelligent interfaces to support cooperating computer users in their interactions with a computer.			
20. DISTRIBUTION/AVAILABILITY OF ABSTRACT <input checked="" type="checkbox"/> UNCLASSIFIED/UNLIMITED <input type="checkbox"/> SAME AS RPT. <input type="checkbox"/> DTIC USERS		21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED	
22a. NAME OF RESPONSIBLE INDIVIDUAL Douglas A. White		22b. TELEPHONE (Include Area Code) (315) 330-3564	22c. OFFICE SYMBOL RADC (COES)

UNCLASSIFIED

Item 10 (SOURCE OF FUNDING NUMBERS) Continued.

Program Element Number	Project Number	Task Number	Work Unit Number
62702F	4594	18	E2
61101F	LDFP	15	C4
61102F	2304	J5	01
33126F	2155	02	10

Item 16. (SUPPLEMENTARY NOTATION) Continued.

This effort was funded partially by the Laboratory Director's Fund.

UNCLASSIFIED

TABLE OF CONTENTS

5

5.1 Executive Summary 5-2

5.2 Introduction 5-3

5.3 Planning and Plan Recognition 5-4

 5.3.1 The POISE System 5-4

 5.3.2 A Semantic Database for POISE 5-5

 5.3.3 The GRAPPLE Project 5-6

 5.3.4 Meta-Plans and First-Principles Knowledge 5-7

 5.3.5 Exception Handling 5-8

5.4 Knowledge Acquisition 5-10

5.5 Cooperative Problem Solving 5-12

 5.5.1 Graphical Aids to Decision Making 5-12

 5.5.2 Intelligent Computer-Aided Instruction 5-15

 5.5.3 Negotiation between Distributed Agents 5-17

- Appendix 5-A Reasoning About Exceptions During Plan Execution Monitoring
- Appendix 5-B Task Management for an Intelligent Interface
- Appendix 5-C Multistage Negotiation in Distributed Planning
- Appendix 5-D Knowledge Acquisition as Knowledge Assimilation
- Appendix 5-E The Role of Plan Recognition in Design of an Intelligent User Interface
- Appendix 5-F Planning Discourse in an Intelligent Tutor
- Appendix 5-G Teaching a Complex Industrial Process
- Appendix 5-H Building a Community Memory for Intelligent Tutoring Systems
- Appendix 5-I A Representation For Collections of Temporal Intervals
- Appendix 5-J A Plan-Oriented Approach to Intelligent Interface Design
- Appendix 5-K Providing Intelligent Assistance in Distributed Office Environments
- Appendix 5-L The GRAPPLE Plan Formalism



Session For	
GRA&I	<input checked="" type="checkbox"/>
TAS	<input type="checkbox"/>
Advanced	<input type="checkbox"/>
Education	<input type="checkbox"/>
Distribution/	
Availability Codes	
Dist	Special
A-1	

Chapter 5

5.1 Executive Summary

The major focus of the 1986 NAIC research at the University of Massachusetts has been development of interfaces that support cooperating computer users in their interactions with a computer. These interfaces have been designed to help people complete tasks and to provide explanations while users engage in their activities. Since knowledge systems must interact with several kinds of users, including programmers, knowledge engineers, experts, end-users, and people learning to be users, the interfaces must be approachable, informative, and capable of communication with users at every level of expertise.

We have been building interfaces that contain both knowledge about typical methods used by people to achieve tasks and knowledge about how to recognize the user's plans. This work has involved research into planning, plan recognition, knowledge acquisition, and cooperative problem solving.

Our work in planning and plan recognition has included significant additions to the POISE interface system and design of an extended new framework called GRAPPLE. Both systems provide a hierarchy of procedural descriptions or plans specifying typical user tasks, goals, and sequences of actions to accomplish goals. POISE represents these plans from an event-based perspective (i.e. sequences of tool invocations) that represent a specific user plan.

The GRAPPLE framework is a second generation intelligent interface. The overriding theme of its plan formalism is an expanded representation of knowledge about plans and their interrelationships. In GRAPPLE, the *goal* of a plan and the *effects* of a plan on the domain model are explicitly represented, as are *preconditions*, which must be satisfied before the plan can be executed. This deeper representation of the plan increases the capacity for reasoning during plan recognition and automated planning and affords the system a much richer knowledge base from which to reason about plan failure. The system also has a larger store of semantic knowledge which it can use to "understand" and accommodate exceptional scenarios during plan recognition.

Among the additions made to POISE, we built a semantic database for representing objects of the domain. This permitted us to represent constraints either from an object or a procedural perspective. POISE used these procedural and object descriptions to recognize the implications of a user's actions and to infer the user's objectives. It used descriptions to plan actions based on the user's goal.

In addition to building these two systems, we developed a multistage negotiation paradigm for planning in a distributed environment. This paradigm was explored in the domain of monitoring and controlling a complex communications system.

As knowledge engineers tire of their labor, they need to build tools to facilitate the transfer of expertise from human expert to expert system. Complex knowledge systems have many hundreds of chunks of knowledge and many kinds of knowledge making control difficult to trace, asynchronous and opportunistic. Knowledge acquisition systems are thus needed to facilitate transfer of both knowledge and control data to large systems.

This year, as part of a Ph.D. dissertation, a system was built to engage an expert in a dialogue about which of several interpretations of knowledge are intended for inclusion into a knowledge base. The knowledge acquisition interface interprets the expert's information and then determines how to include this knowledge into the knowledge base.

This year, we also built intelligent tutoring systems using multiple experts, such as educators, psychologists, physicists and computer scientists, to encode individual teaching and learning expertise. Ideally we would like to define and build frameworks that make changing knowledge simple and that themselves are capable of being rebuilt. This requires a suite of programming tools for browsing and summarizing expert knowledge, for tracing and explaining the student model, and for tracking reasoning about teaching strategies. As a precursor to developing such tools, we have begun implementation of a set of interactive and monitored simulation tutors for teaching college Physics. We expect to develop a framework for building subsequent tutors as a result of our experience with these initial tutors.

In sum, our research this year has focused on problems associated with inferring and anticipating the goals and needs of users, communicating with multiple users, and teaching users expertise that resides in intelligent knowledge bases.

5.2 Introduction

The NAIC AI group at the University of Massachusetts is working in several major areas of intelligent interfaces, including planning, plan recognition, knowledge representation/acquisition, and cooperative problem solving. The major focus of this research is on development of interfaces that support cooperating computer users in their interactions with a computer. These interfaces contain knowledge about typical methods used by people to achieve tasks as well as knowledge about how to recognize plans, help people complete their task, and provide explanations and support to users.

Knowledge systems must interact with several kinds of users, including programmers, knowledge engineers, experts, end-users, and people learning to be users. For a knowledge system to gain acceptance by a community of users it must be approachable and informative. It must understand what the user wants and must rectify misunderstandings gracefully. It must communicate with experts at one level, programmers at another, and amateurs at a third. It must both justify its performance and facilitate its own modification. Preferably such an interface should have command of several communications media, including natural language, programming languages, and graphics, and use these media where they are most appropriate. Needless to say, no current interface is capable of this range of communicative

skill. Still, without working towards such goals AI researchers are building knowledge systems that will be underutilized and not well understood by practitioners in real world domains.

The mechanisms we have built facilitate a user's ability to interact naturally with a system and to improve the computer's ability to describe its own actions and decisions in a clear and user-centered manner. These mechanisms have been developed in the areas of planning and plan recognition, knowledge representation, knowledge acquisition, and cooperative problem solving. This report describes work in each of these areas.

5.3 Planning and Plan Recognition

This year we both completed the POISE interface system [1,2,3] and finished the design of the GRAPPLE [7,8] framework. The POISE (Procedure Orientation Interface for Supportive Environments) interface provides a hierarchy of procedural descriptions or plans that specify typical tasks, goals, and sequences of actions to accomplish goals[4]. This year we built a semantic database for representing objects of the domain to work in parallel with POISE. Given procedural and object descriptions, POISE can recognize the implications of a user's actions and can infer the user's objectives. It can also use these descriptions to plan actions based on the user's goal.

POISE was limited by its reasoning about relationships among plans. The behavioral plan definitions specify decomposition solely in terms of temporally ordered subplans and do not represent plan preconditions or goals. This characterization of a plan is insufficient for a planner whose role is to synthesize new plans, since there is no representation of the reasons behind the substeps, and little knowledge is available about the relationships between different plans. Also, without explicit goals, there is no way to note that an action may be omitted because its goal has already been met. Nor is there enough knowledge to support a robust approach to recognition and recovery from plan failure. With an event-based approach, recovery must be built into the plan rules, making the rules unwieldy and complex and discouraging deeper reasoning about failure and recovery.

As a result of all these limitations, we designed and began implementation of a second generation intelligent interface, GRAPPLE (Goal Recognition and Planning Environment)[5], which extends the previous system primarily through a reformulation of the plans, incorporation of state-based information, and use of meta-plans and reasoning from first-principles. These two systems, POISE and GRAPPLE, are discussed below.

5.3.1 The POISE System

POISE uses predefined, hierarchical activity definitions (plans) to monitor the conversation between user and computer system, recognizing the commands issued as instantiations of primitive plan definitions. Predictions of expected user commands are made as a result of the successful integration of a primitive action into a more abstract plan representation. The interface also operates in an alternate mode to automatically generates sequences of primitive commands in response to user requests for a high-level plan. Such an intelligent interface would thus be a mixed-initiative system, combining facilities for plan recognition and plan

automation, with an embedded planner used to extend the predefined plans as needed.

5.3.2 A Semantic Database for POISE

We completed work on POISE by building an object-based semantic database to augment the procedural knowledge base of the intelligent interface. The model of the user's environment was extended to facilitate interaction with agents external to the interface and to provide the basis for a goal-based mode of operation. A frame-based representation (PRIDE) was developed for the implementation of this semantic database and integrated with POISE.

PRIDE provides a number of predefined facets for specifying meta-level information about slots and their values. These facets are used to mark certain slots as being special, to define constraints on slot values, and to declare triggers (demons) which should be fired upon specified events. All facets are optional. One of the motivating factors for using a custom-designed representation instead of an off-the-shelf tool was to be able to optimize the inheritance mechanism. The notion of inheritance is a useful aid for database design, since it obviates the need for redundant specification, and thus saves memory space. However, the computation time required to retrieve inherited slot values or facets (*e.g.* when checking restrictions on a slot value being added) from even a moderately deep hierarchy of concepts becomes prohibitively expensive. This factor is especially critical for POISE's semantic database, since nearly all accesses are made at the level of object instances. PRIDE was designed to allow the database designer to exploit the notion of inheritance during the static specification of objects.

The development of PRIDE and its integration with the POISE interpretation system raised some interesting issues in knowledge representation concerning the role of the semantic database. In particular, we worried about how to distinguish between the cases in which we should suspend the constraint checking to allow more flexibility and situations in which we should clamp down and claim that an access error has occurred. We also considered the more general AI question represented by the adoption (or rejection) of the "closed-world assumption." The "closed-world assumption" may be summarized as "If you don't have any information allowing you to answer true or false to a statement, assume it is false." The "closed-world assumption" does not recognize "I don't know" as a legitimate response to a request for information.

As a result of these and other considerations, we argued for uniformity in the representation of both procedures and objects and, in fact, to blur the distinction between them. We saw that procedures can be represented as simply another type of object, *e.g.* *procedure-object*, in addition to *world-object*, *relation-object* etc. An examination of the original architecture of POISE against the backdrop of the expanded architecture (with the addition of the semantic database) revealed many parallels between the way that procedures and objects are handled. These parallels furthered the argument for a uniform representation.

Another limitation of POISE was inherent in the way the various types of knowledge were represented. In POISE, domain knowledge was expressed using a frame-like model of domain objects, while the plan knowledge was represented using a completely different formalism and underlying representation tool. The use of a uniform representation for both domain plans and domain objects would allow the system designer to tie together constraints related to both plans and objects, and to provide greater coherency[4]. Another problem was that general

domain knowledge directly related to either a plan or object definition is not represented in POISE. Thus the knowledge needed for a deeper model of the domain is lacking, seriously limiting reasoning that can be done either during plan failure or while handling exceptions that arise during plan recognition.

Given the substantial list of points in favor of representing procedures as special kind of object we decided to include a special kind of object in the object hierarchy for procedures. Procedure attributes would be represented as property slots, as already described for objects. Constraint information contained in the COND clause could be represented as facets on the attributes (slots) of concern and the different sections of the local state could all be represented in this fashion.

5.3.3 The GRAPPLE Project

We designed and began implementation of a second-generation intelligent interface based on hierarchical plans that represent user tasks. Again, recognition of instantiations of these plans occurs by predicting future actions from past events and then matching new actions to these predictions. This new interface, named GRAPPLE, has been used to explore potential sources of deeper domain knowledge than those exploited by POISE, thus motivating a re-evaluation of the characterization and interpretation of plans. As a testbed for GRAPPLE, we used the domain of software development, which is a complex domain offering a rich source of knowledge that is relatively self-contained.

An overriding theme of the GRAPPLE plan formalism is an expanded representation of knowledge about plans and their interrelationships. The *goal* of a plan and the *effects* of a plan on the domain model are explicitly represented, as are *preconditions*, which must be satisfied before the plan can be executed. A deeper representation of the plan increases the capacity for reasoning during plan recognition and automated planning and affords the system a much richer knowledge base from which to reason about plan failure. The system also has a larger store of semantic knowledge which it can use to "understand" and accommodate exceptional scenarios during plan recognition.

The use of a state-based, goal-oriented perspective in GRAPPLE is in contrast to the POISE event-based substep plan characterization and follows the classical planning formalism. A *goal* is specified as a partial state of the semantic database. A goal can be decomposed into *subgoals*, each of which also is expressed as a semantic database state specification. Achievement of all the subgoals, along with the posting of the *effects* of the plan, should lead to satisfaction of the goal of the plan. *Effects* can be expressed in high-level as well as primitive plans, allowing for the expression of complex semantic changes to the semantic database.

A state-based approach to plan representation provides more modularity. For example, if one of the subgoals for a plan is to *have-more-disk-space*, a number of plans may be retrieved that achieve this subgoal; for instance: *delete-a-file*, *purge-directory*, and *increase-quota*. The multiple possible plans need not be specified statically; they can be determined dynamically in order to exploit the rich sources of contextual knowledge at runtime. Representing goals as states in GRAPPLE also allows the interface to avoid a potentially redundant execution of a plan. If a plan has a subgoal which is already satisfied, then no plan need be executed to

achieve the subgoal. The overall ordering of plans which can achieve subgoals of a complex plan is determined dynamically by monitoring the satisfaction of *preconditions*. The state-based approach thus allows for the easy addition and removal of plan definitions from the plan library, without necessitating a recompilation of all the plans and their subgoals. In POISE, the event-based plan specification was "hard-coded," thus rendering the plan library inflexible to dynamic modifications.

GRAPPLE also attempts to overcome limitations imposed by POISE's nonuniform representations. In GRAPPLE, plans are represented with the same knowledge representation tool/language as domain objects. Therefore relationships between certain plans and objects can be easily recorded and constraints relating to both plans and associated domain objects are uniformly specified. The groundwork is laid for a more powerful object representation language and more powerful reasoning capabilities.

In GRAPPLE, an *expected-actions* list is maintained for each top-level plan to record the monitorable user actions predicted by the interface. A *pending-conditions* list is also associated with each top-level plan to record those *goals, subgoals, and preconditions* that await satisfaction.

At any point during the running of the intelligent interface, one or more top-level plans can be in progress. They are represented by instantiations of those plans on the *active plan blackboard*. When a plan is instantiated, each of its goals and subgoals is instantiated as well and maintained as *pending conditions* for that plan. A backward-chaining approach is then taken to predict which plans could achieve these *pending conditions*.

Predictions are currently¹ made by matching the subgoal/goal conditions with the goals of other plans in the plan library. Once a prediction is made, an instantiation structure is created for the predicted plan and its precondition is posted to await satisfaction. If the plan is a primitive one, it is posted to the list of *expected-actions* for the top-level plan that subsumes it.

When a user-action occurs, a *matcher* is invoked to determine which of the *expected-actions* is being performed. Values determined by the filter program, which directly monitors user actions, are passed up to the designated expected action structure, and bindings of variables are propagated. *Pending-conditions* are re-evaluated and the plan recognizer generates new expectations after integrating the action occurrence.

5.3.4 Meta-Plans and First-Principles Knowledge

We are currently working on a meta-plan approach to plan recognition that will provide more relationships between plans in terms of additional subgoal decomposition. Recognizing plan failure and integrating the resulting recovery actions are particularly important in domains like software development, where the basic paradigm of work is "trial and error." Examples of work in the software development environment demonstrate several plan interrelationships. For example, obtaining on-line help provides the user with specific information to formulate and issue additional new commands. Gathering information via tools to analyze, reorganize,

¹A more complete and sophisticated prediction mechanism will be incorporated upon the addition of a more sophisticated *planner* module, which will analyze the interactions between *effects* of plans and pending *goal* conditions.

condense, and present data supports the user in making key decisions about how to carry out some plan. At times, programmers will model a plan with dummy input in order to see if it will work as they predict.

Meta-plans allow us to capture these general patterns as a context for executing any plan, without having to write out all the details in every plan. In our work with meta-plans we have found that the same basic plan formalism with goal, precondition, subgoal, constraints, and effects clauses can be used. Meta-plan variables are not domain objects, rather, they are domain plans, their goals, effects, etc.

As we work with plan definitions in the domain of software development we recognize additional knowledge about the domain that is not appropriately expressed in the plans themselves, such as versions, history, configurations, properties and bugs of modules, and a broad range of *first principles* knowledge about programming. This knowledge forms a self-contained world for reasoning about actions, and will, we believe, be an important addition to the intelligent interface.

This first principles knowledge can be used in the intelligent interface to improve interface performance and extend more assistance to the user. For example, it enables us to generate tentative bindings of plan parameters that result in earlier, more detailed prediction, and also limits the number of alternatives to consider during recognition or execution of plans. It provides an alternative to simple heuristics such as "prefer the continuation of a plan already in progress to the start of a new plan" for choosing among alternatives, which may be increasingly important as the number of alternatives grows or when plans are inherently underspecified. It can be used to double-check decisions made by the programmer. Finally, first principles knowledge can provide additional semantic distinctions between apparently equivalent actions (fixing a bug versus adding a new feature) so that future programmer decisions (such as what tests to run) can be anticipated and double-checked.

We have begun implementing the GRAPPLE plan and semantic database formalism along with plan recognition, constraint handling, and focusing algorithms. Knowledge Craft[6], a knowledge representation tool package that offers a logic programming environment built on top of a frame-based knowledge representation, was used to implement the system. A large set of plans for a Unix²/C software development environment has been written in the GRAPPLE formalism, and we also formalized some first-principles knowledge for this domain[7]. We have started work on defining meta-plans to provide integrated interpretations for the entire spectrum of user actions.

5.3.5 Exception Handling

We have designed and begun implementation of a general architecture to accommodate exceptional occurrences in an interactive planning system[8]. Exceptions are detected by the *execution monitor*. Violations in the plan caused by the introduction of an exception are computed by the *plan critic*. Real-world (not user-generated) exceptions are handled by the *replanner*. The method used to resolve user-generated exceptions depends on the type of exception recognized by the execution monitor. In general, the *reasoner* gathers relevant

²Unix is a trademark of AT & T Bell Laboratories.

information about the relationships between the exception and the expectations.

The *negotiator* determines which agents are affected by the exception and uses the information provided by the reasoner to present suggested changes to the original plan. The negotiator also directs the acquisition of information from the user, if required, again using a trace of the reasoner's search to guide the questioning. The negotiator may invoke the plan critic to detect violations in a plan which remain or result from a proposed change. Negotiation may also be invoked upon the failure of replanning. If the negotiator or replanner produces a consistent explanation of the exception, control is returned to the planner to continue plan execution and generation.

The behavior of the reasoner is guided by general principles derived from the type of the exceptional occurrence. A *step-out-of-order* exception, for example, may imply that the user may be attempting a short-cut, while an *unexpected action* exception may be eventually recognized as an intentional substitution of the unanticipated action for the expected action. The reasoner performs a controlled exploration throughout the knowledge base which is guided by the current state of the procedural network as well as the type of exception which has occurred. If a number of strategies are possible, the least costly is attempted first.

If a user performs an action that doesn't have a match on the *expected-actions list*, the execution monitor first determines whether this action is entirely *unexpected* or is simply *out-of-order*. This determination is made by a search through possible plan expansions. A user action may also be the expected type of action, with an unexpected parameter value (*unexpected parameter*).

The reasoner attempts to establish whether an unexpected action contributes to the pending task in any way. The fundamental assumption is that the unexpected action is related to unachieved goals in the remainder of the plan.

The actual contribution made by this exceptional occurrence can be at an arbitrary level of abstraction and granularity within the task. It may take the place of an expected action, satisfy the precondition of a later action, or eliminate the necessity of an entire sequence of later actions. The effects of the actual action are compared with the preconditions, effects, and goals of other nodes within the procedural net. The reasoner looks for the potential contributions by focusing on the most local contributions first.

When an action is determined to be *out-of-order*, it may be the case that the original ordering may have been specified as a preference, but does not imply strict dependencies between the effects and preconditions of actions. Another possibility is that intervening steps between the expected and actual actions are no longer necessary. In order to determine if either of these cases apply, the reasoner must examine the causal structure of the plan. If the causal structure of the plan is not violated, the exceptional occurrence is allowed. In the second case, an action may be no longer necessary because the goals of the intervening steps may have been accomplished in some "off-line" fashion. The reasoner does nothing in this case, but passes control to the negotiator, which involves the user in an attempt to verify the goals of the intermediate steps.

When an expected action occurs, an unexpected parameter value can cause a constraint violation. Since parameter values are usually objects themselves, the reasoner is invoked to determine what relationships exist between the object provided as the *actual* parameter value

and the object which was *expected* as the parameter value.

Information is collected by the reasoner to establish whether the exceptional parameter should be allowed. The scope of the plan and knowledge base which may be affected by the exception is dependent on the type of constraint violation which has occurred. Modifications and consequences which may result from a static object constraint violation, for example, are localized to the static knowledge base, while plan constraint violations and dynamic object constraint violations may have more far-reaching consequences for the remainder of the plan.

5.4 Knowledge Acquisition

As knowledge engineers tire of their labor, they often build tools to facilitate the transfer of expertise from expert to expert system. Most of these current tools have little research significance: they help the knowledge engineer trace the source of errors in an expert system's performance, then provide a friendly interface for correcting the errors. But knowledge systems are becoming very complex, with many thousands of chunks of knowledge and many kinds of knowledge needing to be represented. Control is often asynchronous and opportunistic, and can be hard to trace. Knowledge acquisition system must be built to facilitate transfer of both knowledge and control data to large systems.

The issue of interpretation underlies all work on knowledge acquisition: The knowledge acquisition system must come to an agreement with an expert about what a piece of knowledge means. This process is called operationalization because it involves translating the acquired knowledge into a form that can be used, operationally, by an interpreter. In general, this is an impossible task because the language in which the initial knowledge is given is ambiguous.

Several difficult research problems must be solved before interpretation of acquired knowledge becomes a straightforward process. Often, knowledge is stated at too high or too low a level of generality, so the knowledge acquisition system must attempt to integrate the new knowledge as an instance (specialization) of old, or cluster several old pieces of knowledge under the new (generalization) knowledge. Additionally, the system must determine which information to revise when new knowledge is inconsistent with the old. It must be able to integrate and reconcile the advice of multiple experts.

Knowledge acquisition requires an understanding of how the new information corresponds to that already known by the system and how this existing information must be modified to reflect the expert's view of the domain. The system discussed below engages an expert in a dialogue about which of several interpretations of knowledge are intended. We have built a system called K^{Ac} , that modifies an existing knowledge base by using heuristic knowledge about the knowledge acquisition process, and by anticipating modifications to the existing entity descriptions[9]. These anticipated modifications, or *expectations*, are used to provide a context in which to assimilate the new domain information.

As the cost of constructing and refining a knowledge base becomes a substantial portion of the cost of constructing an expert system, several approaches to reducing the expense of this labor-intensive task have been examined. Currently, most knowledge bases are built via a series of dialogs between experts in a particular application domain and knowledge engineers familiar with the targeted expert system.

One approach is to provide the knowledge engineer with better tools that improve his efficiency. We proposed techniques to implement guides that automate the assimilation of the expert's knowledge into an existing knowledge base.

An often overlooked aspect of the knowledge acquisition process is the assimilation of information presented by the domain expert into an existing knowledge base. The knowledge engineer's task is to modify knowledge base so as to reflect the domain expert's knowledge. To a large extent, this knowledge acquisition task may be viewed as a recognition problem. All of the problems facing other recognition systems are present here as well, including: noisy data (i.e., incomplete or inaccurate information), ambiguous interpretations, and the need to produce intermediate results before all the data is available. Thus, a significant portion of this interactive knowledge acquisition task is a matching problem: How does the expert's description of the domain correlate with the description contained in the knowledge base? How should the knowledge base be modified based on new information from the expert? What should be done when the expert's description differs from the existing one?

KⁿAc implements this knowledge assimilation approach to knowledge acquisition. It was developed to assist in the construction of knowledge bases for the POISE[1,2,3] intelligent interface system. These knowledge bases used a frame-like representation, described more fully in [2,9] to describe *tasks*, *objects* and *relationships* in the application domain. POISE's initial knowledge bases, for the office automation and software engineering domains, were created by hand from interviews between a knowledge engineer and the appropriate domain experts. Transcriptions of these interviews were examined and the results served as the basis of the KⁿAc system.

It is important to note that the goal of the domain expert was *not* to modify POISE's knowledge base; this was the knowledge engineer's role. The expert simply presented the domain information, e.g., descriptions of tasks, objects, etc., and responded to questions and comments from the knowledge engineer. The burden of assimilating the information, that is, recognizing where it fit into the existing knowledge base and what additions or modifications were needed, was not placed upon the domain expert. (Contrast this to approaches such as [10,11,12]. By modeling the knowledge engineer's role in this task, KⁿAc attempts to provide this same support.

Consider the opening portion of a discourse in which the expert, the principal clerk of an academic department, is describing the procedure for reimbursement for business-related travel expenses.

"O.K. — on travel. The proper way of doing it, if it's out of state, is that a travel authorization should be issued before the trip."

From this information one concludes that some unnamed task consists of two temporally ordered steps. However, it is not clear what modifications need be made to the knowledge base to reflect this information.

If the knowledge base is examined (prior to this interview), a description of this reimbursement process will be found (see Figure 5.1). In this simplified view of the task, which knows nothing about a "travel authorization", the traveler simply goes on a trip and gets reimbursed. Though the knowledge engineer may realize that the clerk and this description


```

EVENT take-a-trip-and-get-paid
STEPS: (take-a-trip get-reimbursed)
TEMPORAL-RELATIONSHIPS:
  (take-a-trip before get-reimbursed)
CONSTRAINTS: ( ... )
ATTRIBUTES: (traveler ...) (cost ...) (destination ...)

```

Figure 5.1: Knowledge Base Event Description

are describing the same task, it is not readily apparent from the two descriptions. Matching such descriptions, and recognizing the implied modifications, are central to the assimilation process.

To accomplish this, K^{Ac} was required to perform two basic tasks: 1) recognizing where the expert's information fits into the existing knowledge base, and 2) appropriately modifying the existing knowledge so that it reflects the expert's view of the domain. Determining where the expert's information fits into the existing knowledge requires that the new information be matched against the existing information. To avoid matching the new information against the entire (existing) knowledge base, the most likely candidate matches should be selected. Furthermore, since the goal of a knowledge acquisition discourse is modification of the knowledge base, exact matches between the new and the existing information are not always expected.

The K^{Ac} System is based on a procedure for matching the expert's entity descriptions with those already in the knowledge base. It recognizes that discrepancies between the two descriptions may imply that modifications are needed especially if the discrepancies (or the implied modifications) can be predicted. K^{Ac} performs these predictions based on anticipated modifications, or *expectations*, that arise from an understanding of the knowledge acquisition process. These predictions can be derived from the state of the existing knowledge base, from cues in the discourse, from previous modifications to entity descriptions, or from the state of the knowledge acquisition task.

5.5 Cooperative Problem Solving

Several projects have been developed to explore issues in cooperative problem solving or the exchange of reasoning and knowledge as a part of communication between intelligent agents. In each case, we have designed a system to assist intelligent agents in making decisions or in learning new material based on information gleaned from another agents. Three such projects are described below.

5.5.1 Graphical Aids to Decision Making

Decision support problems can either be thoroughly structured, unstructured, or a combination of the two. Well-structured problems, such as inventory reordering, are recurring

problems that are largely clerical in nature. For these recurring clerical problems there is a wealth of clerical computer programs. For less structured problems such as financial planning, upgrade of manufacturing facilities, R & D budgeting, or architectural design, there are no comprehensive solution by computer programs. Programs may assist, but the principal part of these problems are beyond simple computational techniques. For planning problems such as these, a technique is needed for modeling a system, exploring the sensitivities of the model to alternative designs, and analyzing the results with respect to the various goals. ThinkerToy is designed as such a system.

ThinkerToy is a graphical environment for modeling decision support problems. It provides a tableau on which problems, such as landscape planning, service scheduling, and statistical analysis can be modeled and analyzed. It uses graphical icons each with associated physical properties to replace mathematical relationships and properties. The key construct in this methodology is the ManiplIcon, an icon that is not just pictorial in nature, but also a semantic tool for building models that homomorphically represent semi-structured problems.

ThinkerToy is a homogeneous object oriented system where every object is a graphical entity and is directly manipulable. Together these objects create a language whose grammatical rules are formed by the constituents and whose semantics and syntax are revealed by the visual metaphors it employs. This homogeneity extends from the very lowest to the very highest parts of the system:

Scalars: Tools for Trig functions, Log functions, detection, injection, and applying values.

Arrays: Tools for ripping out, injecting, and overlaying values on the face of tabular data.

Charts: Tools for shape fitting, axis stretching, and extraction nets.

TerrainMaps: Tools for physical and pseudo-physical molding and growing of features on terrain and thematic maps.

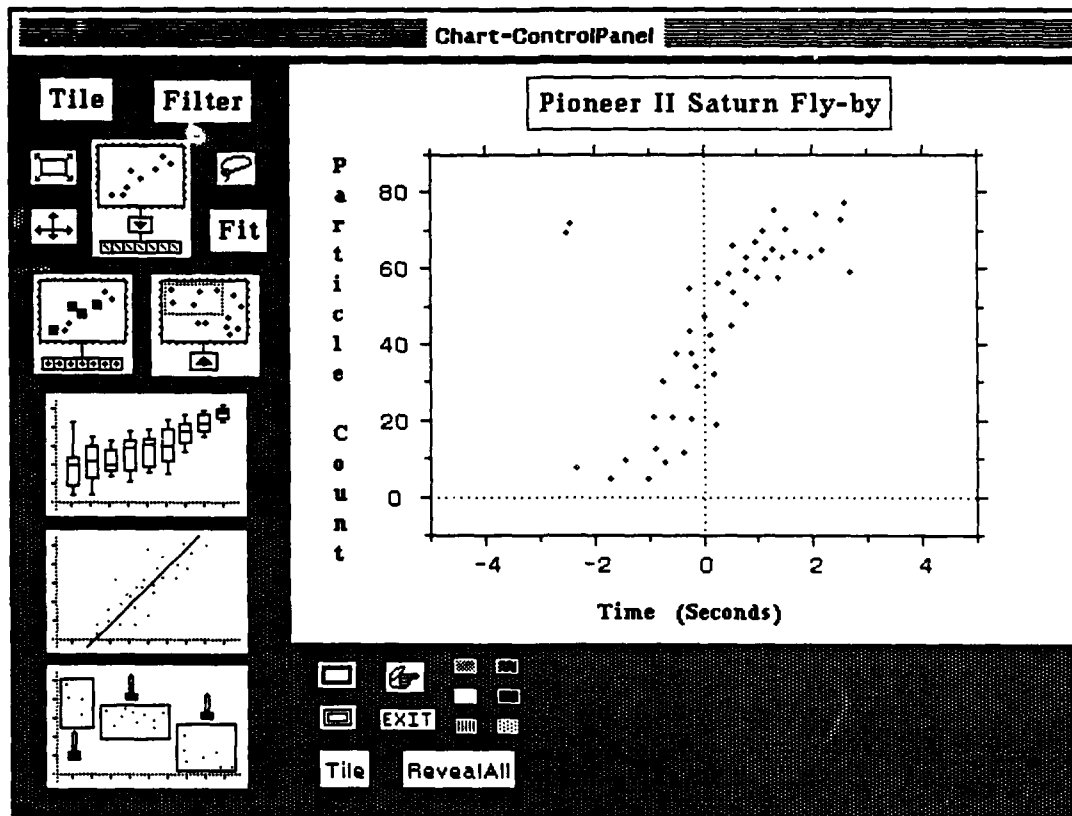


Figure 5.2: Chart

For example, figure 5.2 show a second two-dimensional object (chart) and the panel that is used to manipulate its objects. Similar screens exist for one and three-dimensional figures. The key construct for accomplishing homogeneity within a single dimension are Panels (also referred to as ManipIcons). A ManipIcon is an active icon whose actions are invoked via manipulating it with a mouse. There are Panels that represent a broad set of the basic Smalltalk object family: Integer, Float, Symbol, Array, Form. There are also composite Panels that represent tools useful in building tableaus: Scales, Charts, Thermostats, Buttons, ControlBoards, ControlPanels, Toolkits, etc.

The ThinkerToy implementation kernel is meant to form the foundation for constructing larger systems. Interesting applications begin to occur when users create a FlowModel, instrument it with charts as meters, collect data from experiments, use the chart to perform statistical analysis, and then perform iterative changes via arrays. Only when one begins to use all components together and then produces hybrid models from this base, does the power of the ThinkerToy environment become apparent.

5.5.2 Intelligent Computer-Aided Instruction

One of the most valuable skills an intelligent interface can have is the ability to teach its users. Unfortunately, Intelligent Computer Aided Instructional (ICAI) systems typically require enormous resources to design, implement, and evaluate.

Building such systems requires community knowledge, i.e., multiple experts working together to encode individual expertise during a knowledge acquisition phase that might span months or years [13]. These experts require a framework that will make *changing knowledge simple* and that will itself be *capable of being rebuilt*. They require a suite of programming tools for browsing and summarizing their knowledge, for tracing and explaining the student model, and for tacking reasoning about teaching strategies [14]. In short, tools and methodologies are needed for knowledge acquisition in an intelligent tutor. This year we have worked on building tools to facilitate such an acquisition process [15].

In addition, knowledge needed to build an intelligent tutor is often distributed, erroneous, and acquired incrementally [16]. This is especially true because the domain expert, cognitive scientist, and teaching expert are not typically the same person. One of the goals of our work is to make the knowledge contributed by each expert both modular and explicit. In this way, multiple experts can work together to integrate their own knowledge and that of others into the Expert System.

We are addressing this design bottleneck by developing graphic browsing facilities that allow experts to interact easily with tutoring systems in the service of incremental knowledge acquisition. The browsers will enable domain experts, teachers, psychologists, and computer scientists to visualize the structure of existing concepts and the relationships among concepts and rules. The goal of these browsers is to make it possible for a variety of experts to create, modify, or delete concepts and rules in the knowledge base. Currently, tutoring systems have been built for specific applications, thus allowing the building process to remain a black art—a pre-technology which requires a great deal of experimentation and effort to produce minimal results. A modular framework will make this knowledge acquisition process more reasonable.

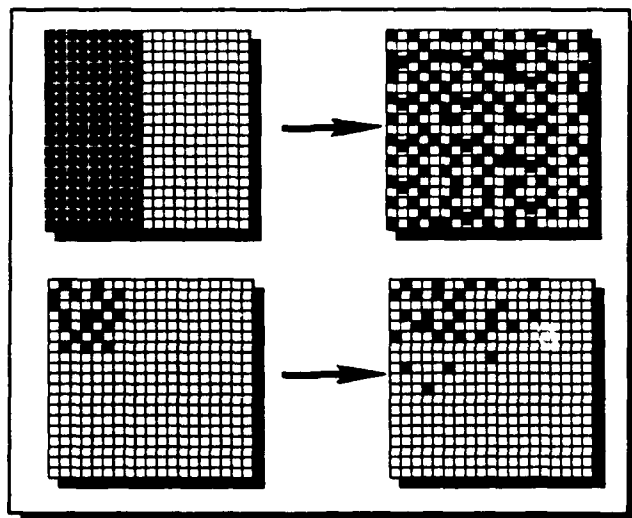


Figure 5.2: Systems Moving Towards Equilibrium

We are now building simulation tutors for physics using a team of experts including physicists, teachers, and computer scientists. These tutors are designed to put a student in direct contact with physics elements, such as atoms, energy, and heat. The student can use a variety of activities, such as changing the position or velocity of a body in a celestial mechanics simulation to view dependent changes in the size, speed, and position of the orbit. The tutor monitors and advises the student while she/he works, it provides examples, analogies, or explanations based on the student's actions, questions, or responses.

The first system we have built teaches about the second law of thermodynamics.³ It is taught at the atomic level [17] and provides a rich environment through which the principles of equilibrium, entropy, and thermal diffusion can be observed and tested. The student is shown, and is able to construct, collections of atoms that can transfer heat to each other through random collision, Figure 5.2.

The student creates areas of "high" energy atoms, indicated by dark squares, along with variously-shaped observation regions to analyze and monitor the energy. Several systems can be constructed, each with monitorable areas of high energy. Concepts such as temperature, energy density, and thermal equilibrium for each system can be plotted against each other and against time. Thermodynamic principles, such as heat transfer through random collision and entropy as a function of the initial organization of the system, can be observed.

At any time the student can modify the temperature of the system, the number of collisions per unit time, and the shape of the observation regions. All student activities, along with questions, responses, and requests, are used by the tutor to formulate its next teaching goal and next activity. We are now building the knowledge base of tutoring strategies to

³The second law states that heat can not be absorbed from a reservoir and completely converted into mechanical work.

reason about whether to show an extreme example, or a near-miss one, to give an analogy, or to ask a question. We are also studying student misconceptions and common errors in learning thermodynamics and statistics to refine the tutor's response.

5.5.3 Negotiation between Distributed Agents

We have focused on developing a multistage negotiation paradigm for planning in a distributed environment[18]. The application domain involves the monitoring and control of a complex communications system. The multistage negotiation protocol is useful for cooperatively resolving resource allocation conflicts which arise in a distributed network of semi-autonomous problem solving nodes. The primary contributions of such a negotiation protocol are that it makes possible the detection and resolution of subgoal interactions in a distributed environment with limited communication bandwidth and no single locus of control. Furthermore, it permits a distributed problem solving system to detect when it is operating in an overconstrained situation and to remedy the situation by reaching a satisficing solution.

The distributed environment in which our negotiation takes place is a network of loosely coupled problem solving agents in which no agent has a complete and accurate view of the state of the network. Problem solving activity is initiated through the instantiation of one or more top level goals at agents in the network. Each top level goal is instantiated locally at an agent and is not necessarily known to other agents. Since the conditions which give rise to goal instantiation may be observed at more than one place in the network, the same goal may be instantiated by two or more agents independently. The desired solution to the problem is any one that satisfies all of the top level goals.

Multistage negotiation has been devised as a paradigm for cooperation among agents attempting to solve a planning problem in this distributed environment. One of the major difficulties that arises is detecting the presence of subgoal interactions and determining the impact of those interactions. In distributed applications, the problem is exacerbated because no agent has complete knowledge concerning all goals and subgoals present in the problem solving system. For example, subgoals initiated by one node may interact with other subgoals initiated elsewhere, unknown to the first node. These interactions may become quite complex and may not be visible to any single node in the network. A second issue that arises in planning is recognizing when goals are not attainable.

When a node begins its planning activity, it has knowledge of a set of top level goals which have been locally instantiated. A space of plans to satisfy each of these goals is formulated during plan generation without regard for any subgoal interaction problems. After plan generation, each node is aware of two kinds of goals: *primary goals* (or p-goals) and *secondary goals* (or s-goals). In our application, p-goals are those instantiated locally by an agent in response to an observed outage of a circuit for which the agent has primary responsibility (because the circuit terminates in the agent's subregion). These are of enhanced importance to this agent because they relate to system goals which *must* be satisfied by this particular agent if they are to be satisfied at all. An agent's s-goals are those which have been instantiated as a result of a contract with some other agent. An agent regards each of its s-goals as a possible alternative to be utilized in satisfaction of some other agent's p-goal.

A plan commitment phase involving multistage negotiation is initiated next. As this phase begins, each node has knowledge about all of the p-goals and s-goals it has instantiated. Relative to each of its goals, it knows a number of alternatives for goal satisfaction. An alternative is comprised of a local plan fragment, points of interaction with other agents (relative to that plan fragment), and a measure of the cost of the alternative (to be used in making heuristic decisions). Three characteristics of distributed planning problems motivate development of a more general cooperation paradigm. First, subgoal interaction problems which arise in the context of a distributed planning system when agents do not have a global view are very difficult to detect and even more difficult to handle in a reasonable way. Second, many application domains embody planning problems that are overconstrained. When these planning problems are addressed by a network of planning agents, it is essential that the system be able to determine whether or not the problem is overconstrained. Third, when the planning problem is overconstrained, it is necessary for the agents involved to arrive at an agreement as to a set of goals whose satisfaction is regarded as an acceptable solution to the problem at hand. Each of these issues without the re-exchange of sufficient knowledge as to permit each agent to construct a global view.

Bibliography

- [1] Croft, W.B., Lefkowitz, L., Lesser, V., and Huff, K. "POISE: An Intelligent Assistant for Profession-based Systems," in *Proceedings of the Conference on Artificial Intelligence*, Oakland University, Michigan, April 1982.
- [2] Croft, W.B. and Lefkowitz, L.S. "Task Support in an Office System," *ACM Transactions on Office Information Systems*, vol. 2, pp. 197-212, 1984.
- [3] Carver, N., Lesser, V. and McCue, D. "Focusing in Plan Recognition," *Proceedings of National Conference on Artificial Intelligence (AAAI-84)*, Austin, Texas, pp. 42-48, 1984.
- [4] Broverman, C.A. and Croft, W.B. "A Knowledge-based Approach to Data Management for Intelligent User Interfaces," *Proceedings of Conference for Very Large Data Bases 11*, Stockholm, Sweden, 1985, pp. 96-104.
- [5] Broverman, C.A., Huff, K.E., and Lesser, V.R. "The Role of Plan Recognition in Design of an Intelligent User Interface," *Proceedings of IEEE Conference on Systems, Man and Cybernetics*, pp. 863-868, 1986.
- [6] Knowledge Craft Manual Guide, Vax/VMS Version 3.0, Carnegie Group Inc., March 1986.
- [7] Huff, K.E. and Lesser, V.R., "Intelligent Assistance for the Process of Programming," Technical Report 87-09.
- [8] Broverman, C. and Croft, W. B., "Reasoning About Exceptions During Plan Execution Monitoring," *Proceedings of National Conference on Artificial Intelligence (AAAI-87)*.
- [9] Lefkowitz, L.S. and Lesser, V.R. "Knowledge Acquisition as Knowledge Assimilation," University of Massachusetts, Amherst, Department of Computer and Information Science Technical Report 87-13.
- [10] Eshelman, L., Ehret, D., and McDermott, J. "MOLE: A Tenacious Knowledge Acquisition Tool," *Proceedings of the Knowledge Acquisition for Knowledge-Based Systems Workshop*, Banff, Alberta, Canada, 1986.

- [11] Ginsberg, A., Weiss, S., and Politakis, P. "SEEK2: A Generalized Approach to Automatic Knowledge Base Refinement," *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pp. 367-374, 1985.
- [12] Kahn, G., Nowlan, S., and McDermott, J. "MORE: An Intelligent Knowledge Acquisition Tool," *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pp. 581-584, 1985.
- [13] Woolf, B., and Cunningham, P. "Building a Community Memory for Intelligent Tutoring Systems," *Proceedings of the National Conference on Artificial Intelligence (AAAI-84)*, Morgan Kaufmann, Inc., Los Altos, CA, August 1984.
- [14] Woolf, B., and McDonald, D. "Context-dependent Transitions in Tutoring Discourse," in *Proceedings of the National Conference on Artificial Intelligence (AAAI-84)*, Morgan Kaufmann, Inc., Los Altos, CA, August 1984.
- [15] Woolf, B., Blegen, D., Verloop, A., and Jensen, J. "Tutoring a Complex Industrial Process," *Proceedings of the National Conference on Artificial Intelligence (AAAI-84)*, Morgan Kaufmann, Inc., Los Altos, CA, August 1986.
- [16] Bobrow, D., Mittal, S., and Stefik, M. "Expert Systems: Perils and Promise," *Communications of the ACM*, September 1986.
- [17] Atkins, T.; *The Second Law*, Freedman, San Francisco, CA, 1982.
- [18] Conry, S.E., Meyer, R.A., and Lesser, V.R. "Multistage Negotiation in Distributed Planning," University of Massachusetts, Amherst, Department of Computer and Information Science Technical Report 86-87.

Reasoning About Exceptions During Plan Execution Monitoring*

Carol A. Broverman
W. Bruce Croft
Department of Computer and Information Science
University of Massachusetts
Amherst, Massachusetts 01003
(413) 545-0463
broverman@cs.umass.edu
croft@cs.umass.edu

June 24, 1987

Paper-type: short paper

Topic Area: Reasoning

Track: Science

Keywords: planning, execution monitoring, exception handling

Abstract

In a cooperative problem-solving environment, such as an office, a hierarchical planner can be incorporated into an intelligent interface to accomplish tasks. During plan execution monitoring, user actions may be inconsistent with system expectations. In this paper, we present an approach towards reasoning about these *exceptions* in an attempt to accommodate them into an evolving plan. We propose a representation for plans and domain objects that facilitates reasoning about exceptions.

*This work is supported by the Air Force Systems Command, Rome Air Development Center, Griffiss Air Force Base, New York 13441-5700, the Air Force Office of Scientific Research, Bolling Air Force Base, District of Columbia 20332, under contract F30602-85-C-0008, and by a contract with Ing. C. Olivetti & C.

1 Interactive planning and exceptional occurrences

Hierarchical planners incrementally develop a plan at different levels of abstraction, imposing linear orderings at each stage of the expansion to eliminate subgoal interactions [8,9,11]. The execution of the plan's primitive actions must be monitored to ensure success. Exceptions and interruptions are common occurrences, and the planner must react to new information made available during the various stages of plan construction and execution. Existing plans may require modification or new plans may have to be generated.

We are concerned with using a planner as a support tool in a cooperative problem-solving environment such as an office [2,4]. In such an environment, the planner is not viewed as an omnipotent agent with complete knowledge of the domain and procedures for accomplishing all plan steps. Rather, it aids the user in performing correct and consistent tasks. The operation of the planner depends heavily on interaction with the user in order to allow user control and to draw on the users' domain knowledge. Interactive planners necessarily interleave plan generation and execution since user actions determine the course of future events.

Previous planners have provided general replanning actions which are invoked in response to problems in the plan resulting from the introduction of an arbitrary state predicate or "fact" [6,8,12]. In these systems, the replanning techniques provided do not attempt to reason about failing conditions or possible serendipitous effects of the exception. These methods simply make use of the explicitly linked plan rationale to detect problems and determine what violated goals need to be reacheived. We view this type of replanning as a "reactionary" tactic involving little intelligence, and reserve its use for exceptions generated by *external agents*¹.

To address the problems associated with interactive planning, we propose extending the traditional replanning approach. When a user action deviates from the planner's predictions, the system should exploit available knowledge in an attempt to explain the exceptional behavior. Such a constructive approach is preferred to replanning, since replanning, in this case, would attempt to achieve goals that the user deliberately chose not to pursue. This paper discusses *reasoning about exceptional occurrences* as an approach towards incorporating exceptions into a consistent plan. In the next two sections, we describe an interactive planner and the elements of our representation which are used to support the reasoning process. We then outline the types of exceptions that can occur and algorithms for handling them, within the context of an example taken from the domain of real estate.

2 An interactive planner

Input to our interactive planner is provided as an abstract goal specification, and the output is a partially or fully expanded procedural net, with partial temporal ordering (similar to other hierarchical planners [8,9,11]). A procedural net contains goal nodes, action nodes, and phantom nodes (goal nodes which are trivially true), along with links representing the causal structure of

¹The planner attempts to satisfy a number of *agents*. The user(s) are regarded as *internal agents*, while agents are considered to be *external* if the system lacks a model for their behavior (e.g., the *real world*).

the plan. Since complete expansion of the initial goal may require additional information from the user, only action nodes are considered primitive, and thus executable.

We distinguish between those primitive action nodes which the system is able to carry out using available tools (*system-executable*) and those which must be executed by the user (*user-executable*). An action node may be both system-executable and user-executable, in which case automation is preferred. An example of an action which may be solely user-executable could be the cancellation of an order; the decision to cancel must be initiated by the user and thus can be modeled as a decision action occurring "offline" [3]. Transferring information from a purchase request to an order form, however, is a primitive action which may either be performed by the user or automated.

At any point during the planning and execution of a task, an *expected-action list* contains the set of user-executable primitive actions which are not preceded by unexpanded goal nodes. This is the set of actions which are predicted by the system to occur next. As each system-executable or user-executable action is performed, the procedural net is expanded further, producing an updated expected-actions list. A user action may be inconsistent with system expectations, in which case it is flagged as an *exceptional occurrence*.

3 A representation for plans and domain objects

An important part of our approach is a uniform object-based representation of *activities, objects, agents and relationships*² [2]. An integrated abstraction hierarchy (see Figure 2) combined with a powerful constraint language facilitates the representation and use of more sophisticated knowledge about plans, such as the *policies* of McDermott [7]. The reasoning process described in the next section exploits this object-based representation. A similar approach has been used by Alterman [1] and Tenenbergs [10] to represent old plans that are adapted to new situations.

The major features of our representation are a *taxonomic knowledge, aggregation, decomposition, resources, plan rationale and relationships*. Each of these is defined and illustrated using an example from the domain of house-purchasing, shown in Figures 1 and 2. Figure 1 depicts a partially expanded procedural net fragment which represents the portion of a house-buying task which remains after a house has been selected for purchase. Figure 2 shows a portion of the domain knowledge relevant to this task.

Any complex entity can be viewed as a composition of several other objects as well as an aggregation of properties. An abstract activity object which can be decomposed into more detailed substeps has a *steps* property containing a partial ordering of more detailed activity steps. Decomposition of a domain object into other objects is expressed as a set of object types named in a *parts* property. The aggregation of all properties of either an activity or domain object, including decomposition information, constitutes the object definition.

All entities are represented in a type hierarchy, with inheritance along *is-a* links between types and their subtypes. Entities inherit the properties and constraints of their supertypes. For example, a *mortgage-application-form* has various fields which are inherited from the more general *form* object, and obeys the constraint stating that it can be manipulated by an *apply* type of activity

²In the remainder of the paper, we refer to plan descriptions as *activities* and objects of the domain simply as *objects*.

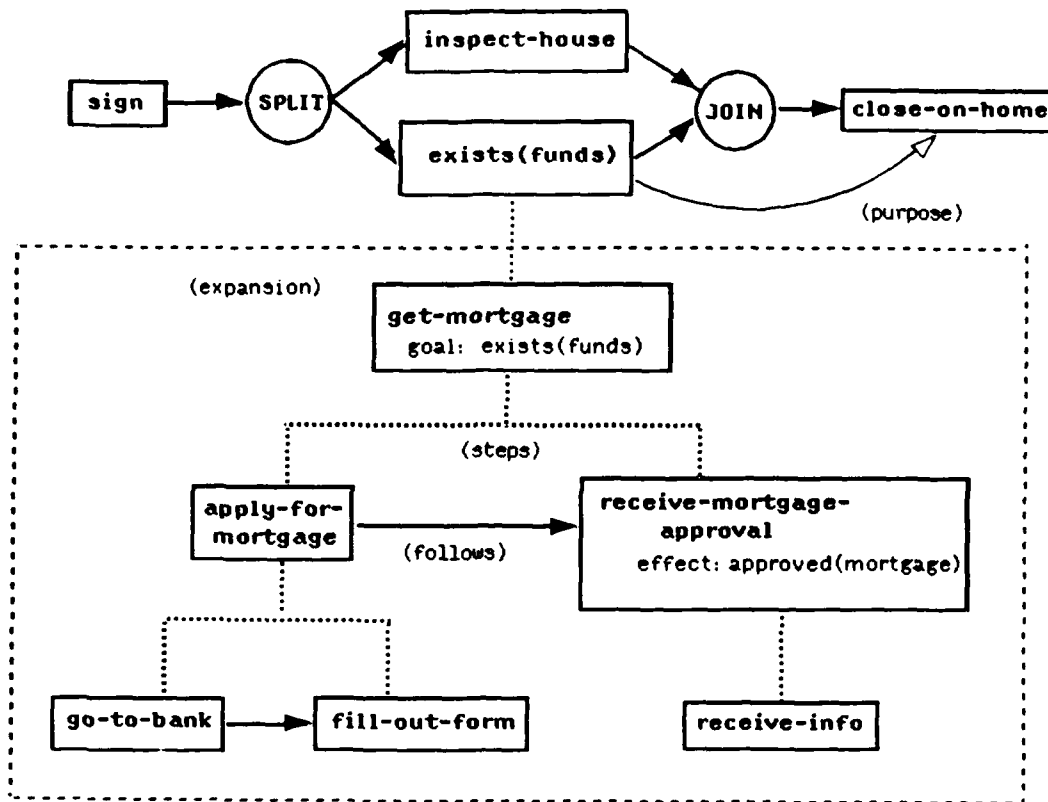


Figure 1: Example procedural net fragment

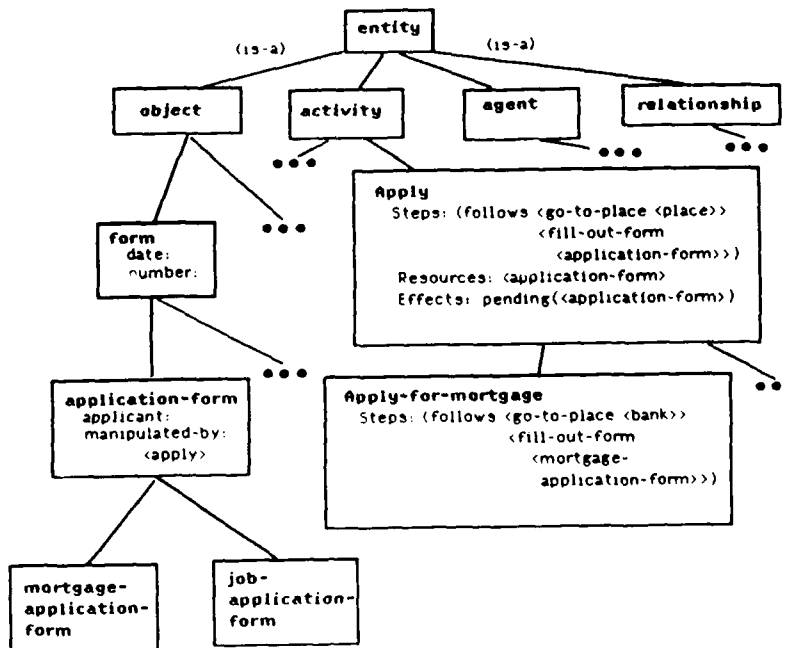


Figure 2: Fragment of knowledge base

(inherited from *application-form*). Activities inherit the preconditions and effects of their super-types, as well as decomposition information. For example, any apply activity may be decomposed into an activity of type *go-to-place* followed by *fill-out-form*. *Apply-for-mortgage* is a subtype of *apply* and thus inherits and specializes this decomposition. *Apply-for-mortgage* also inherits the effect of *pending(application-form)*.

An activity has an associated set of *effects* which are asserted upon its completion. Effects are represented as predicates on domain objects. The *goal* of the activity is a distinguished main effect and is used for matching during plan expansion. An activity schema also includes a declaration of the types of domain objects it may manipulate. The inverse of this *resources* property is the *manipulated-by* property expressed in domain objects to indicate which types of activities may affect them. The union of an activity schema with the descriptions of associated object types provides a rich semantic representation of the domain, incorporating objects and operators.

Causal knowledge is represented by *goal* properties and *purpose* links. *Goals* are of a global nature, in that they relate an activity to a representation of its intent; that is, they state what this activity accomplishes regardless of the context of the current procedural net. *Purpose* links may be placed between two plan substep nodes in both static and dynamic plan representations, to indicate that a substep of a plan produces a state required for the proper execution of a later substep, much like NONLIN's goal structure [9]. The *purpose* links prove to be particularly important in determining whether or not an exception can easily be incorporated into an existing plan.

Arbitrary *relationships* may also exist between domain objects. For example, a *seller* relation may be depicted between an individual and a certain *house*, expressing the fact that someone is selling a particular house. A special type of relationship which may exist between two objects is a *transformation* relation, which contains a procedural attachment for producing the correct instance of one type of object associated with the instance of the second object type. For example, the abstract class object *address* may be related to *telephone-number* through a special transformation specification which indicates that a phone call using a *phone-number* may produce the corresponding *address*.

4 Unexpected occurrences

A user action occurs within the context of predictions made by the system. Exceptions can be generated by unanticipated user actions. Because of the inherent open-endedness of the domain, an unexpected occurrence may in fact be a valid semantic action, not recognized as such because of an inaccurate or incomplete activity description.

Referring back to our example depicted in Figures 1-2, we can imagine the following possible scenarios:

- (a) Suppose *receive-mortgage-approval* has occurred. We are expecting an *inspect-house* action by the user. Instead, the user executes the first step of the *close-on-home* procedure, *go-to-closing-location*. This is an instance of a *step-out-of-order* exception, since this step is expected, but not until later in the plan.
- (b) Suppose the *purchase-and-sale-agreement* has been signed, and the system next expects the

user to start carrying out the steps to obtain a mortgage (*go-to-bank*). Instead, a *sell-stock* action is taken by the user, generating an *unexpected-action* exception.

- (c) Suppose that while the user is waiting for his mortgage to be approved, his friend from the bank stops in the office and hands him a hard-copy of the approval. Since the normal way of receiving approval is in the form of an electronic message, the user simply offers a *user-assertion* by introducing the predicate *approved(mortgage)*.
- (d) Suppose, that while executing the *fill-out-form* substep of the *apply-for-mortgage* step, the user fills in the address field with a *phone-number* instead of an *address*, triggering a constraint violation. This is a case of an *expected action, unexpected parameter* type of exception, where a static object constraint violation has occurred. Unexpected parameters can result in violations of other types of constraints, such as a static constraint in the activity schema, or a constraint dynamically posted on a domain-object by an activity instance.

The above scenarios illustrate the classes of unexpected occurrences which can arise. Actions can be *out-of-order* or completely *unexpected*. A *user-assertion* arbitrarily introduced to the system may have implications for the current plan. A user assertion is modeled as an unexpected action with the assertion as its main effect, and is treated as an unexpected action. An expected action may occur with an *unexpected parameter*, resulting in the violation of a static or dynamically posted object constraint, or the violation of a constraint within the plan itself. In the following sections, we develop algorithms for reasoning about the various types of exceptions, and show how each of the above scenarios can be resolved, resulting in a consistent plan.

5 A general architecture for exception handling

While this paper focuses primarily on the reasoning process used to handle exceptions, a general architecture designed to accommodate exceptional occurrences is shown in Figure 3. Several of the modules are similar to those described in other hierarchical planners, specifically [12]. We have extended the basic replanning model to include additional modules (highlighted in Figure 3) to address exception handling. Exceptions are detected by the *execution monitor* and classified by the *exception classifier*. Violations in the plan caused by the introduction of an exception are computed by the *plan critic*. Real-world (not user-generated) exceptions are handled by the *replanner*. The replanning approach we have adopted is similar to that of [12], where one or more of a set of general replanning actions is invoked in response to a particular type of problem introduced into a plan by an exceptional occurrence. For interactive planning, we extend the set of general replanning actions to include the insertion of a new goal into the plan.

The *exception analyst* applies available domain knowledge in an attempt to construct an explanation of an exception. Its primary function is to determine the relationships and compatibility of the actual events to the expected actions, goals and parameters. The particular entity relationships investigated by the exception analyst are determined by the type of internal exception. The exception analyst may be triggered by both external and internal exceptions, although it is primarily used for internal exceptions.

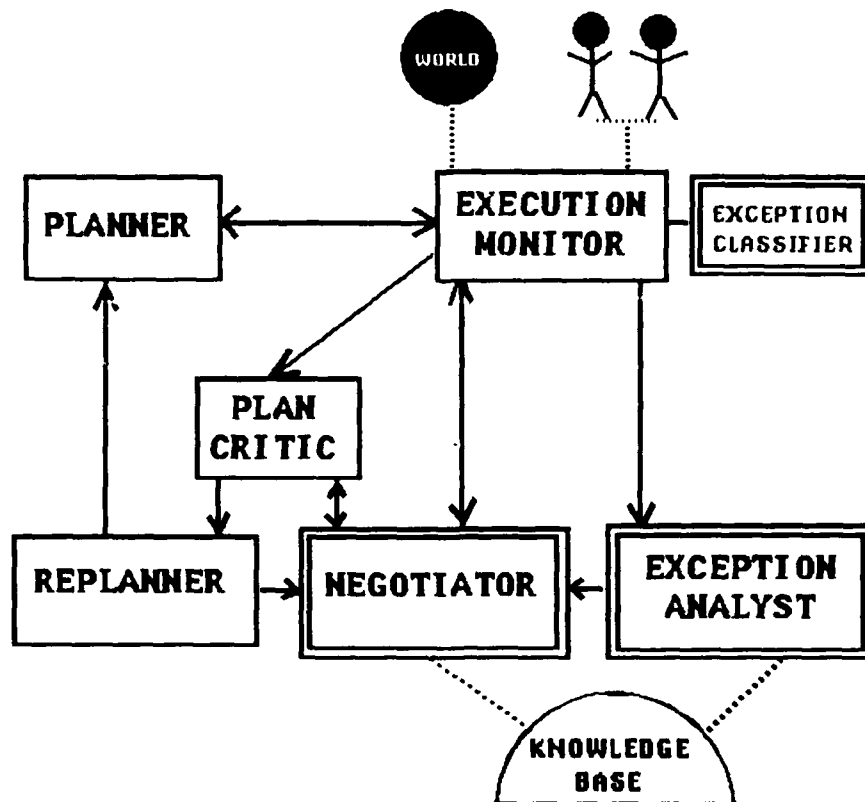


Figure 3: An architecture for a cooperative planner

The paradigm of *negotiation* [5] has been used as a model for reaching an agreement among agents on a method for accomplishing a task. We propose to use negotiation for establishing a consensus among agents who are affected by an exception. The *negotiator* determines the set of affected agents and uses the information provided by the exception analyst to present suggested changes to the original plan.

We distinguish between *effecting* and *affected* agents with regard to the occurrence of an exception. The *effecting* agent is that agent who has caused the exception. An *affected* agent is one whose interests are influenced (either positively or negatively) by the exception. Affected agents are those who are "responsible" for the parts of the plan where problems are detected by the plan critic. An external agent can never be an *affected* agent, since the system has no model of an external agent's interests or behavior.

Using information provided by the exception analyst about relationships between actual and expected values, the negotiator initiates an exchange between the effecting agent and the affected agents. The negotiator and plan critic execute in a loop in which the plan critic analyzes changes suggested by the negotiator to detect any problems introduced. This loop is exited when no further problems are detected by the plan critic and all affected agents are satisfied.

The negotiator also directs the acquisition of information from the user, if required, again using a trace of the exception analyst's search to guide the questioning. Negotiation may also be invoked upon the failure of replanning. If the negotiator or replanner produces a consistent explanation of the exception, control is returned to the planner to continue plan execution and generation. A successful negotiation can result in a system which has "learned," that is, the static domain plans may be augmented with knowledge about the exception and thus enhances the system's capability to handle future similar exceptions.

6 Reasoning about exceptions

The behavior of the exception analyst is guided by some general principles derived from the type of the exceptional occurrence. A *step-out-of-order* exception, for example, may imply that the user may be attempting a short-cut, while an *unexpected action* exception may be eventually recognized as an intentional substitution of the unanticipated action for the expected action. The exception analyst performs a controlled exploration throughout the knowledge base which is guided by the current state of the procedural network as well as the type of exception which has occurred. If a number of strategies are possible, the least costly is attempted first. In the following sections, we present algorithms for handling the various types of exceptions, illustrating (where relevant) with the example scenarios developed in section 3.

6.1 When the action taken doesn't match an expected one

If a user performs an action which doesn't have a match on the *expected-actions list*, the exception classifier is invoked to determine whether this action is entirely *unexpected* or is simply *out-of-order*. This determination is made by a search through possible plan expansions.

6.1.1 Unexpected action

If a user action occurs which is not expected anywhere in the plan, the exception analyst attempts to establish whether this unexpected action contributes to the pending task in any way. The fundamental assumption is that the unexpected action is related to unachieved goals in the remainder of the plan.

The unexpected action may be related to the expected action or to another plan step which is predicted later in the plan expansion. The actual contribution made by this exceptional occurrence can be at an arbitrary level of abstraction and granularity within the task. In other words, an action may take the place of an expected action, satisfy the precondition of a later action, or eliminate the necessity of an entire sequence of later actions. The effects of the actual action are compared with the preconditions, effects, and goals of other nodes within the procedural net. The exception analyst looks for the potential contributions by focusing on the most local contributions first. The control of the exception analyst is illustrated by the following algorithm:

1. Can the exceptional action be *substituted* for an expected action? If either of the following criteria are met, a substitution should be allowed:
 - (a) Effects of the exceptional action *exactly* match those of the expected action.

Scenario (c) is an example where a *user-assertion* is introduced to inform the system of the results of an actions which has occurred "offline." The exception analyst notes that the effects of *receive-mortgage-approval* are matched by this dummy action, making the expected action no longer necessary.
 - (b) The intersection of effects of the exceptional and expected actions are exactly those effects of the expected action which have *purpose* links to later plan steps.

2. Does the exceptional action allow a *simplification* of the remainder of the plan?

(a) If the action can be substituted for a *later* step in the plan (established by the above method), treat the exception as an *out-of-order* action (below) and record the substitution of the matching actions.

(b) Do any of the effects of the exceptional action match with an unachieved effect which is the *purpose* for a later plan step? If so, a later precondition is satisfied; note that the precondition is now a phantom, but do not modify expectations.

3. Does the unexpected action allow an entire hierarchical wedge to be removed from the plan?

If the exceptional action results in the satisfaction of a higher-level goal, the steps comprising the expansion of that goal may no longer be necessary. The exception analyst determines the parent node of the expected action. If the goal of this parent node is achieved by the effects of the exceptional action, then the following is done: Check to see if the effects of each of this parent's children (excluding exceptional action itself) are now true. If none of the unachieved effects have *purpose* links to steps occurring after the parent node, then a substitution is allowed. The exceptional node is incorporated in the procedural net, and the expected action, its parent and siblings are considered to be achieved.

This method can be applied to scenario (b). The exception analyst notes that the exceptional step *sell-stock* has the same goal (*exists(funds)*) as a more abstract step in the plan expansion, namely *get-mortgage*. The user may intend to buy the house with his own funds, and not the bank's. The hierarchical wedge of the plan which constitutes the expansion of *get-mortgage* is removed from the plan and replaced by *sell-stock*.

6.1.2 Out-of-order action

If the action is judged to be an out-of-order plan step, there are two possibilities to consider:

1. The original ordering may have been specified as a preference, but there are no strict dependencies between the effects and preconditions of actions. In order to determine if this is the case, the exception analyst must examine the causal structure of the plan. Specifically, if there are no *purpose* links between the actual step and an intervening step which has not been performed, the ordering may be relaxed.

This case applies to scenario (a). The exception analyst notes that the *inspect-house* action is optional, since there are no *purpose* links from that node to nodes later in the plan. Therefore, a relaxation of the originally specified ordering is allowed.

2. The intervening steps between the expected and actual actions are no longer necessary. This may be because the goals of the intervening steps may have been accomplished in some "offline" fashion. The exception analyst does nothing in this case, but passes control to the negotiator, which involves the user in an attempt to verify the goals of the intermediate steps.

6.2 Unexpected parameter exceptions

When an expected action occurs, an unexpected parameter value can cause a constraint violation. Since parameter values are usually objects themselves, the exception analyst is invoked to determine what relationships exist between the object provided as the *actual* parameter value and the object which was *expected* as the parameter value. The exception analyst attempts to establish the following:

1. The two objects may have a *common ancestor* in the object hierarchy. If so, the exception analyst constructs the set of features *unique* to the expected object, since the lack of these features in the object actually provided as the parameter value may be problematic.
2. The two objects may both be *manipulated-by* activities which belong to a common activity superclass. If so, they probably are utilized in similar fashions.
3. There may be any number of other *relationships* between the two objects. Specifically, a *transformation* relationship may link the object provided with the expected object, describing a method to obtain the expected parameter value.

To handle scenario (d), the exception analyst notes that the *phone-number* object and *address* objects are linked through a *transformation* relationship, specifying that a procedure *call* may be used on the phone number to produce the corresponding address.

4. The discrepancy between the two parameters may result from *differing quantities* of the object type. If so, an excess may or may not be allowable. The semantics associated with the underlying data type are particularly important when handling quantity discrepancies, since commonsense reasoning may be required. For example, if the *go-to-bank* step was supposed to result in withdrawing 50 *dollars*, emerging with 100 may not be problematic, but baking a cake in a 450 *degree* oven when the recipe calls for 350 degrees may have unsatisfactory results.

This information collected by the exception analyst is used during *negotiation* to establish whether the exceptional parameter should be allowed. The scope of the knowledge base which may be affected by the exception is dependent on the type of constraint violation which has occurred. Modifications and consequences which may result from a static object constraint violation, for example, are localized to the static knowledge base, while plan constraint violations and dynamic object constraint violations may have more far-reaching consequences for the remainder of the plan.

7 Status

Implementation of a prototype which incorporates the ideas presented in this paper is currently underway. One of the major aims of this work is to augment domain plans with knowledge acquired during exception handling. We are currently looking at the issue of propagating change in an object-based representation.

References

- [1] Alterman, R. "An adaptive planner", *Proceedings of AAAI-86*, 65-69, 1986.
- [2] Broverman, C.; Croft, W.B. "A knowledge-based approach to data management for intelligent user interfaces", *Proceedings of VLDB 11*, Stockholm, 96-104, 1985.
- [3] Broverman, C.A., Huff, K.F., Lesser, V.R. "The role of plan recognition in design of an intelligent user interface", *Proceedings of IEEE Conference on Man, Machine, and Cybernetics*, 863-868, 1986.
- [4] Croft, W.B.; Lefkowitz, L.S. "Task support in an office system", *ACM Transactions on Office Information Systems*, 2: 197-212; 1984.
- [5] Fikes, R.E. "A commitment-based framework for describing informal cooperative work", *Cognitive Science*, 6: 331-347; 1982.
- [6] Hayes, P.J. "A representation for robot plans", *Proceedings IJCAI-75*, 181-188, 1975.
- [7] McDermott, D.V. "Planning and Acting", *Cognitive Science*, 2, 1978.
- [8] Sacerdoti, E.D. *A Structure for Plans and Behavior*, Elsevier North-Holland, Inc., New York, NY, 1977.
- [9] Tate, A. "Generating project networks", *Proceedings IJCAI-77*, Boston, 888-893, 1977.
- [10] Tenenber, J. "Planning with Abstraction", *Proceedings of AAAI-86*, 76-80, 1986.
- [11] Wilkins, D.E. "Domain-independent planning: Representation and plan generation", *Artificial Intelligence*, 22: 269-301; 1984.
- [12] Wilkins, D.E. "Recovering from execution errors in SIPE", SRI International Technical Report 346, 1985.

Task Management for an Intelligent Interface

W. Bruce Croft

University of Massachusetts

ABSTRACT

An intelligent interface assists users in the execution of their tasks. To do this, the system must be able to represent tasks and the objects that are manipulated. The intelligent interface described in this paper uses an object management system to manage object and task instantiations and the relationships between them. The object management system is viewed as an implementation of a data model that emphasizes the modeling of operations.

1. INTRODUCTION

In interactive computing environments, the users play a dominant role in determining the operation of the system by selecting the services or tools that are required for their tasks. A task is simply a sequence of activities, some of which are performed on the computer, which taken together accomplish users' goals. Examples of this type of environment are office information systems, software development environments and CAD/CAM systems. In such systems, the interaction with the user is typically viewed as an unpredictable series of tool invocations, rather than as the execution of tasks which are at a higher level of abstraction. The lack of knowledge of user tasks severely limits the role of the system during the interaction. To address this limitation, we define an *intelligent interface* as a subsystem that provides a means of describing and supporting the typical interactions users have with the computing environment. The primary function of the intelligent interface is to provide a wide range of assistance to users in the execution of their tasks.

The characterization of a user's interaction with a system presents a number of problems that cannot be addressed with conventional programming languages. The following features of task description are particularly important:

1. Tasks involve user actions as well as executable code. Often they are nondeterministic.
2. Tasks must be able to be specified by users with widely varying computer experience.
3. Task descriptions are often incomplete. The description of a task must be able to change as the user's understanding of the task changes.
4. Task descriptions represent only typical actions involved in carrying out a task. Exceptions to these typical patterns are very common.

This research was supported in part by the Rome Air Development Center and by Digital Equipment Corporation.
Author's address: Computer and Information Science Department, University of Massachusetts, Amherst, MA 01003 (413/545-0463).
CSNET: *croft@umass*.

The POISE system [CROF84] was designed to address the problems of task definition and support. In this system, tasks are specified as underconstrained plans [COHE82, Ch. 15]. A task is described in terms of subtasks, associated objects, local variables, the preconditions for the task and the effect of carrying out the task. It is underconstrained in the sense that the exact ordering of subtasks is often not specified or only partially specified. The primitive tasks in a task hierarchy are either the operations provided by the tools or application programs. No further breakdown of these operations is necessary to execute them. Not all of the lowest-level tasks in a task hierarchy need be primitive tasks; they may currently only be specified at an abstract level or they may correspond to actions that occur outside the system (e.g., making a telephone call).

As the user specifies more details about a task, or as the system learns more about a task, the task descriptions are further constrained by the addition of rules that affect the ordering of subtasks or the relationships of objects or variables used by subtasks. New subtasks representing more detailed actions may be added. Examples of these added constraints are

- a rule specifying when step A must come before step B
- a rule specifying that the object used in step B is the same as the object in step D.

In this way, the system builds up detailed plans for tasks that are initially specified at a higher level of abstraction by the users.

The system uses the task descriptions to predict user actions (as well as automating aspects of the task). When an *exception* to the predicted action occurs, the system is alerted to the fact that its task description is inadequate and it can then take appropriate action. The emphasis on acquiring knowledge through exceptions is also found in Borgida's work [BORG85]. Many types of exceptions can occur including, for example, different orderings of subtasks, missing subtasks, subtasks activated with preconditions not satisfied, and object constraint violations.

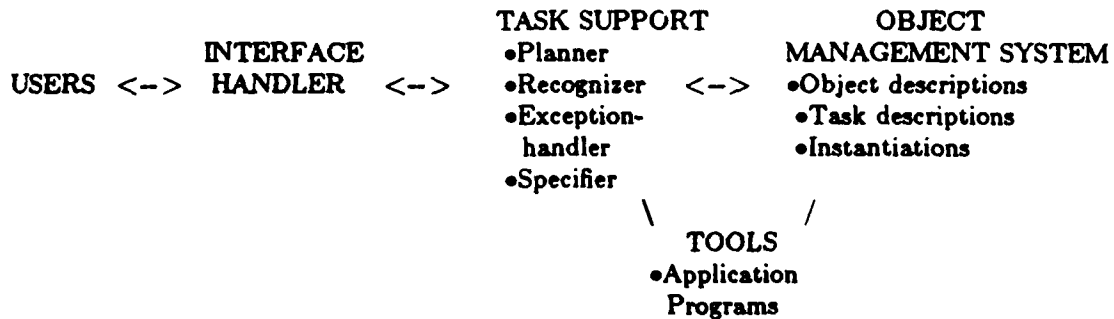


Fig. 1. The POISE system.

The basic architecture of the system incorporating task definition and support is shown in Figure 1. The interface handler is responsible for presenting to the users an integrated view of the tasks, tools, and objects that are available. Users can invoke tasks or manipulate objects directly with the tools. The task support module "understands" the user actions and choices, records them, and takes appropriate actions. This module has four major components. The *planner* executes plans (task descriptions). This includes predicting user actions and propagating constraints from one task step to another. The *recognizer* is used to recognize plans that the user is following without having been specifically invoked. This includes the recognition of exceptions. Recognition of plans in ambiguous situations requires sophisticated control and backtracking mechanisms [CARV84]. The *exception handler* is used to update task descriptions in response to specific user actions. The *specifier* provides the means for users to specify tasks. This specification is done through a graphical interface and requires the user to describe tasks in terms of subtasks, relationships between subtasks, and objects that are manipulated.

The object management system provides facilities for describing objects and tasks and for managing

their instantiations. Tools can be viewed as a special class of application program that manipulates the objects stored in the object management system. For example, in an office system, the tools would include an editor, a forms package, a spreadsheet, and a mail facility. In this paper, we shall describe how the object management system can be considered to be an implementation of an extended data model.

TASK: Purchasing

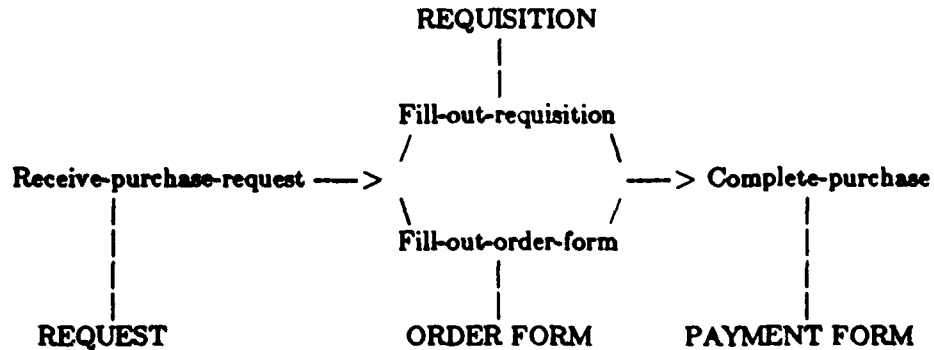


Fig. 2. An example task.

A simple example of the operation of this system in the office environment is given by the purchasing task shown in Figure 2. This shows the task at the highest level of abstraction. The description of the purchasing task, its subtasks and the associated objects (such as the order form) reside in the object management system. The task description contains a constraint that a request for a purchase must occur before an order form or a requisition can be filled out. It also specifies that either one of those steps must occur before completing the purchase. The other form of constraint relates the contents of the request, the order form and the payment form. At this level of abstraction, the task description will look very similar to an ICN specification [ELLI82]. Some of the steps in the task description will be specified at a greater level of detail. For example, the "Fill-out-order-form" subtask may contain a detailed description of how this step is accomplished. Other steps, such as "Fill-out-requisition", may be only partially specified. It is the responsibility of the task support module to monitor the user's interactions with the system, recognize when a requisition is being used and to gather information that will further specify this step. Once the purchasing task has been specified by the user, it is presented by the interface handler as one of the "tools" available to the user. When a particular purchase is required, the user would invoke this task and the system would create instantiations of the purchasing task and related objects such as the order form.

2. DATA MODELS AND EXTENSIONS

Data models provide a means of defining the structure of objects in a particular environment, constraints on those objects, and operations that may be performed on them [TSIC82]. Much of the research in this area has concentrated on the static aspect of object description, rather than the dynamic aspect. To support the intelligent interface, however, we are forced to look at the task descriptions and ask how they are related to application programs, transactions and the data manipulation languages provided in conventional database systems. We define an extended data model as consisting of a means to describe objects, a means of describing operations and a means of describing the connections between objects and operations. Constraints are specified as part of both the object and operation definitions.

Object definitions are accomplished using a data model such as that described in Gibbs [GIBB84], which allows non-first normal form objects, generalization hierarchies and constraints defined using domain specifications and trigger procedures. For example, in an office application, an order form that

contains a variable number of ordered items may be defined as a specialization of a general form object. The order form may inherit a constraint from the general form object that the form number should be between 1 and 99999. A specific constraint, that the total field should be the sum of the costs of the items, could also be defined.

The operations that can be defined include tasks, application programs, tools and transactions. The primitive operations, which are provided in the data manipulation language in database systems, are predefined and apply to all objects. These operations include creating, updating, deleting and retrieving objects. A containment hierarchy of operations, as shown in Figure 3, results from the observation that operations higher in the hierarchy are described in terms of operations that are lower in the hierarchy.

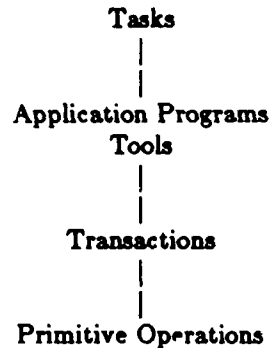


Fig. 3. Operation hierarchy.

A distinction can be drawn between atomic and non-atomic operations. The primitive operations and transactions are atomic in the sense that they are indivisible from the user's point of view. On the other hand, the steps involved in tasks and application programs can be visible to the users and may require user input. Task concurrency and constraint checking thus cannot be handled in the same manner as transactions. Delaying constraint checking until the end of a task, for example, is not possible because the intermediate states are visible to the users. The fact that tasks can be suspended indefinitely also requires that locking does not occur as it would for a transaction. These points lead to the conclusion that transactions can only be defined for the very low level operations from the user's point of view. The maintenance and checking of task instantiations in order to provide a consistent view of the system's operation to the user is entirely the responsibility of the task support module. For example, the task support module can assist the user in "undoing" the steps of a task and can check constraints whenever new information becomes available.

The main advantages of introducing task, application program, and tool operations into the data model are that the connections between user-level operations and objects can be made explicit and a common framework is provided for describing and managing the static and dynamic aspects of a system. Generalization hierarchies of operations, multiple instantiations of operations, and inheritance of operations through specialization of object types can all be described. For example, it is possible to describe a "Fill-out-form" task that is connected with a general form object. We could then describe a "Fill-out-order-form" task as a specialization of the more general task that includes more steps and constraints. An order form, which is a specialization of the general form object, would have a connection to the "Fill-out-order-form" task but would also inherit operations connected to the the general form, such as "Get-form-number".

In contrast to the Smalltalk view [GOLD83], where objects are defined through the operations that are attached to the object, our view is that the operations and the structure of the objects are both of interest and have separate descriptions, but are tightly connected (a kind of "marriage of equals"). An alternative description of the extended data model, which is more object-oriented in nature, would view tasks and objects as two subclasses of a more general object class. Operations that are attached directly to

objects are atomic whereas task "objects" describe user-level operations that typically are non-atomic and manipulate a number of other objects. The extended data model is closely related to the model described by Stemple and Sheard [STEM82, SHEA85].

The description of the operations vary according to the operation type. Task descriptions were mentioned in the last section. The programming languages and data manipulation languages used to describe application programs and transactions in conventional database systems are the major part of the description of these operations, but other information is needed. From the point of view of the intelligent interface, the most important information about these operations is the name, the functionality, and the input/output characteristics. That is, given a task step, the POISE system has to know what lower-level operations can carry out that step, how these operations can be invoked, and what information is required.

As mentioned previously, the description of operations involves a definition of constraints. These constraints, either explicitly defined in task descriptions, or implicit in the application program code, define allowable *transitions* of the object instantiations and the operation instantiations. It has been recognized that static and transition constraints are not independent and that redundant specifications are not uncommon [SHEA85]. POISE is designed to use either form of a constraint during planning and recognition. For example, a task description constraint may specify that if an order amount is less than \$500, the step "Fill-out-order-form" is appropriate, otherwise "Fill-out-requisition" should be used. In the description of objects, the same constraint could be specified by allowing only values less than \$500 in the amount field of the order form. By allowing users to specify this constraint in either way, POISE simplifies the task description process.

3. OTHER MANAGEMENT ISSUES

A number of other problems arise in the management of the object and operation instantiations for the intelligent interface. One of these is that in this type of system it is essential to know which people or more accurately, "agents", can carry out tasks. The description of agents and the "roles" that they take has been the subject of previous research [ELLI82]. In the system described in this paper, agents would be represented as a class of objects with connections to both tasks and other objects.

During the process of planning and recognition, the intelligent interface must keep track of assumptions that are made in order to backtrack should a mistake be made or if the users change their actions. Part of this record keeping involves version histories of the objects [ZDON84]. However, in the intelligent interface, histories of operation instantiations are also required. This situation is further complicated by the fact that there may be multiple interpretations of a single user action, only one of which may turn out to be valid. The process of planning also requires the propagation of constraints into "predicted" versions of the objects. The interpretations in this system are similar to *contexts* used in some systems developed for artificial intelligence research [BARR82, p. 35].

By representing operations and objects in a single framework, the management problem is considerably simplified. A task instantiation can have a set of object instantiations associated with it. These object instantiations can be either "base" objects or "constraint" objects. Base objects record the state of the objects as seen by the users. Constraint objects are used as placeholders for propagating constraints and making predictions. The definition of a constraint object is a "relaxed" version of the base object definition. For example, a particular field in a base object may be specified as containing an integer in the range 1 to 100. The constraint object version of the field has to be able to hold values such as " $20 < x < 60$ " to allow for symbolic propagation of constraints.

The object management system is partially implemented using a frame-based language [WRIG83]. At this level, both the operations and objects are represented as *frames*. Facilities such as generalization hierarchies and triggers are typically supported in these languages. The *slots* of the frames can hold any type of information, including code, and can therefore be used for the complex datatypes and constraints used in the extended data model. The planner and recognizer have previously been implemented as independent modules and are currently being reimplemented to take advantage of the object management

system.

ACKNOWLEDGMENTS

The author benefited from many discussions with David Stemple. POISE was designed jointly with Victor Lesser.

REFERENCES

- [BARR82] Barr, A. and Feigenbaum, E.A, eds., *The Handbook of Artificial Intelligence, Vol. 2*, William Kaufmann, Los Altos, CA, 1982.
- [BORG85] Borgida, A. and Williamson, K., "Accommodating exceptions in databases and refining the schema by learning from them", *Proc. 11th Int. Conf. on Very Large Data Bases*, 1985, pp. 72-81.
- [CARV84] Carver, N.F., Lesser, V.R., and McCue, D.L., "Focusing in plan recognition", *Proc. AAAI Conf.*, 1984, pp. 42-48.
- [COHE82] Cohen, P. and Feigenbaum, E.A., eds., *The Handbook of Artificial Intelligence, Vol. 3*, William Kaufmann, Los Altos, CA, 1982.
- [CROF84] Croft, W.B. and Lefkowitz, L.S., "Task support in an office system", *ACM Trans. on Office Information Systems* 2(3), 1984, pp. 197-212.
- [ELLI82] Ellis, C.A. and Bernal, M., "Officetalk-D: an experimental office information system", *Proc. ACM SIGOA Conf. on Office Information Systems*, 1982, pp. 131-140.
- [GIBB84] Gibbs, S., *An Object-Oriented Office Data Model*, Ph.D. Thesis, Dept. of Comp. Sc., Univ. of Toronto, Toronto, Canada, 1984.
- [GOLD83] Goldberg, A. and Robson, D., *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, Reading, MA, 1983.
- [SHEA85] Sheard, T., *Proving the Consistency of Database Transactions*, Ph.D. Thesis, Comp. and Inf. Sc. Dept., Univ. of Massachusetts, Amherst, MA, 1985.
- [STEM82] Stemple, D., *Generalized Type Specifications for Database Systems*, Tech. Rep. 82-15, Comp. and Inf. Sc. Dept., Univ. of Massachusetts, Amherst, MA, 1982.
- [TSIC82] Tsichritzis, D.C. and Lochovsky, F.H., *Data Models*, Prentice-Hall, Englewood Cliffs, NJ, 1982.
- [WRIG83] Wright, M. and Fox, M.S., *SRL 1.5 User Manual*. Intelligent Systems Lab., Robotics Inst., Carnegie-Mellon Univ., Pittsburgh, PA, 1983.
- [ZDON84] Zdonick, S., "Object management system concepts", *Proc. ACM SIGOA Conf. on Office Information Systems*, 1984, pp. 13-19.

APPENDIX 5-C

Multistage Negotiation in Distributed Planning

S.E. Conry
R.A. Meyer
V.R. Lesser

December 15, 1986
Coins Technical Report 86-67

This work was supported, in part, by the Air Force Systems Command, Rome Air Development Center, Griffiss Air Force Base, New York 13441-5700, and the Air Force Office of Scientific Research, Bolling AFB, DC 20332 under Contract No. F30602-85-C-0008. This contract supports the Northeast Artificial Intelligence Consortium (NAIC). This work was also supported, in part, by the National Science Foundation under CER Grant DCR-8500332, and by the Defense Advanced Research Projects Agency (DOD), monitored by the Office of Naval Research under Contract NR049-041. This work was done while S.E. Conry and R. A. Meyer were on sabbatical at the University of Massachusetts.

Multistage Negotiation in Distributed Planning

S. E. Conry and R. A. Meyer
Clarkson University
Potsdam, N. Y. 13676
(315) 268-6510, -6541, -6511
conry @ clvms (bitnet)

V. R. Lesser
University of Massachusetts
Amherst, MA 01003
(413) 545-1322
lesser @ umass-cs.umass.edu (csnet)

December 15, 1986

Paper Type: Full-paper

Topic: Reasoning

Track: Science

Keywords: Distributed problem solving

Abstract

In this paper we describe a multistage negotiation paradigm for planning in a distributed environment with decentralized control and limited interagent communication. The application domain of interest involves the monitoring and control of a complex communications system. In this domain planning for service restoral is performed in the context of incomplete and possibly invalid information which may be updated dynamically during the course of planning. In addition, the goal of the planning activity may not be achievable — the problem may be overconstrained. Through multistage negotiation, a planner is able to recognize when the problem is overconstrained and to find a solution to an acceptable related problem under these conditions. A key element in this process is the ability to detect subgoal interactions in a distributed environment and reason about their impact. Multistage negotiation provides a means by which an agent can acquire enough knowledge to reason about the impact of local activity on nonlocal state and modify its behavior accordingly.

This work was supported, in part, by the Air Force Systems Command, Rome Air Development Center, Griffiss Air Force Base, New York 13441-5700, and the Air Force Office of Scientific Research, Bolling AFB, DC 20332 under Contract No. F30602-85-C-0008. This contract supports the Northeast Artificial Intelligence Consortium (NAIC). This work was also supported, in part, by the National Science Foundation under CER Grant DCR-8500332, and by the Defense Advanced Research Projects Agency (DOD), monitored by the Office of Naval Research under Contract NR049-041. This work was done while S. E. Conry and R. A. Meyer were on sabbatical at the University of Massachusetts.

1 INTRODUCTION

We present a multistage negotiation protocol that is useful for cooperatively resolving resource allocation conflicts which arise in a distributed network of semi-autonomous problem solving nodes. The primary contributions of such a negotiation protocol are that it makes it possible to detect and to resolve subgoal interactions in a distributed environment with limited communication bandwidth and no single locus of control. Furthermore, it permits a distributed problem solving system to detect when it is operating in an overconstrained situation and act to remedy the situation by reaching a satisficing [1] solution.

Multistage negotiation is specifically *not* intended as a mechanism for goal decomposition in the system, though some goal decomposition is a natural result of negotiation in the context of this paradigm. Our protocol may be viewed as a generalization of the contract net protocol [2,3,4]. The contract net was devised as a mechanism for accomplishing task distribution among agents in a distributed problem solving system. Task distribution takes place through a negotiation process involving contractor task announcement followed by bids from competing subcontractors and finally announcement of awards. Multistage negotiation generalizes this protocol by recognizing the need to iteratively exchange inferences drawn by an agent about the impact of its own choice of what local tasks to perform in satisfaction of *global goals*.

Multistage negotiation produces a cooperation strategy similar in character to the Functionally Accurate/Cooperative paradigm [5] in which agents iteratively exchange tentative and high level partial results of their local subtasks. This strategy results in solutions which are incrementally constructed to converge on a set of complete local solutions which are globally consistent. Before describing multistage negotiation in detail, we first motivate the need for a new cooperation paradigm.

2 MOTIVATION FOR MULTISTAGE NEGOTIATION

The distributed environment in which our negotiation takes place is a network of loosely coupled problem solving agents in which no agent has a complete and accurate view of the state of the network. Problem solving activity is initiated through the instantiation of one or more top level goals at agents in the network. Each top level goal is instantiated locally at an agent and is not necessarily known to other agents. Since the conditions which give rise to goal instantiation may be observed at more than one place in the network, the same goal may be instantiated by two or more agents independently. The desired solution to the problem is any one that satisfies all of the top level goals.

In this type of distributed network, it is very expensive to provide a complete global view to each agent in the system. Communication bandwidth is generally limited. Exchange of enough information to permit each agent to construct and maintain its own accurate global view would be prohibitively expensive. In addition, progress in problem solving would be significantly slower due to a decrease in parallelism attributable to the need for synchronization in building a complete view. Multistage negotiation has been devised as a paradigm for cooperation among agents attempting to solve a planning problem in this distributed environment. In the remainder of this section, we explain the contributions of

multistage negotiation in solving distributed planning problems.

One of the major difficulties which arises in planning systems is detecting the presence of subgoal interactions and determining the impact of those interactions. In distributed applications, the problem is exacerbated because no agent has complete knowledge concerning all goals and subgoals present in the problem solving system. For example, subgoals initiated by one node may interact with other subgoals initiated elsewhere, unknown to the first node. These interactions may become quite complex and may not be visible to any single node in the network. *A key objective of our multistage negotiation is to allow nodes to exchange sufficient information so that these interactions are detected and handled in a reasonable manner.* This objective is achieved by exchanging knowledge about the nonlocal impact of an agent's proposed local action without requiring the exchange of detailed local state information.

Another significant issue that arises in planning is recognizing when goals are not attainable. When satisfaction of a goal requires the commitment of resources, conflicts may arise among goals competing for limited resources. A planning problem is overconstrained if satisfaction of one top level goal precludes the satisfaction of others. Detection of an overconstrained situation in a distributed environment is, again, particularly difficult because no agent is aware of all goals, and each agent has only a limited view of the complete set of conflicts. *When a number of alternative choices for goal satisfaction are known, detection of an overconstrained situation is not possible without either multistage negotiation or a global view.*

In an overconstrained problem, a planning system must reformulate what it seeks as a satisfactory solution. Having several equally important top level goals, the planner must decide which ones should be sacrificed to permit satisfaction of others. Since the distributed network has no agent with sufficient knowledge to serve as an intelligent arbitrator, a consensus must be reached. *Multistage negotiation provides a mechanism for reaching a consensus among those nodes with conflicting goals concerning an acceptable satisfying solution.*

In the following sections, we first describe the problem in more detail, discussing the application domain itself as well as an example which illustrates the nature of the planning problem. We then discuss two models of problem solving relevant to this domain: one which is oriented from the perspective of a single goal and one which is node centered. In the fifth section we discuss a multistage negotiation protocol which utilizes these models and has been incorporated in a distributed planner for this problem. We illustrate this protocol with the aid of a simple example. Finally, we discuss ways in which this research extends existing work.

3 APPLICATION DOMAIN

The application domain of interest is the monitoring and control of a complex communications system. This system consists of a network of sites, each containing a variety of communications equipment, interconnected by links. These sites are partitioned into several geographic subregions with a single site in each subregion designated as a control facility. Each control facility has responsibility for communication system monitoring and control

within its own subregion and corresponds to a single node in the distributed problem solving network. In order to distinguish between the communication network and the problem solving network, in this paper we reserve the term "site" to mean a physical location in the communication system. The term "node" will be used to refer to those sites at which processing and control reside.

The communication network considered here represents a long-haul, transmission "backbone" of a larger, more complex communications system. From this transmission oriented perspective, each user is provided with a dedicated set of resources (equipment and link bandwidth) which establishes a point-to-point connection, or circuit for a significant period of time. Any equipment failure or outage will cause an interruption of service to one or more users.

An overall knowledge-based system to perform the monitoring and control function would employ distributed problem solving agents involving data interpretation, situation assessment, fault diagnosis, and planning [6]. In this context planning is used to find restoral plans for user circuits which have been interrupted as a result of some failure or outage.

A restoral plan consists of a logical sequence of control actions which allocate scarce resources in order to restore end-to-end user connectivity (circuits). These actions allocate or reallocate equipment and link capacity along some route to specific circuits and are subject to a number of constraints. For example, a circuit is assigned to one of several priority categories. In attempting to restore service, resources belonging to circuits of a lower priority may be preempted. Depending upon the type of circuit, there may be special equipment needs which are not necessarily present at all sites. Available routes through the network may be constrained by lack of certain equipment items such as switches or multiplexers. Thus generation of a restoral plan for a single circuit uses conventional route finding algorithms [7] in combination with knowledge about circuit types and priority, needed equipment, network topology, and equipment configuration at all sites along the restoral path. For any specific circuit there will generally be many alternative restoral plans, so the planning system must then attempt to select a combination of alternatives which restores all circuits.

There are a number of features of this planning problem that make it interesting. There is implicit in this domain the assumption that the knowledge of each agent is incomplete. It may also be inaccurate and inconsistent with that of other agents. Restoral plans must be generated in a distributed fashion because no agent has a global view and reliability issues mitigate against delegating the responsibility for planning to a central node. The overall system goal is one of determining plans for restoral of all interrupted service. Although each agent implicitly knows this goal, it generally will not know all of the specific circuits which require restoral. The planning system need not satisfy the overall goal to be successful. In many instances, the overall goal may be infeasible, and thus a satisfactory plan will fall short of reaching this goal.

The distributed planning problem addressed in this paper and our approach to solving it can best be understood with the aid of an example. A simplified diagram of a small network is shown in Figure 1. In this phase of our work, we use a simplified model of a communications system which disregards any constraints arising from equipment configuration at a site. There are five subregions, labeled A, B, C, D, and E, shown. Each site is

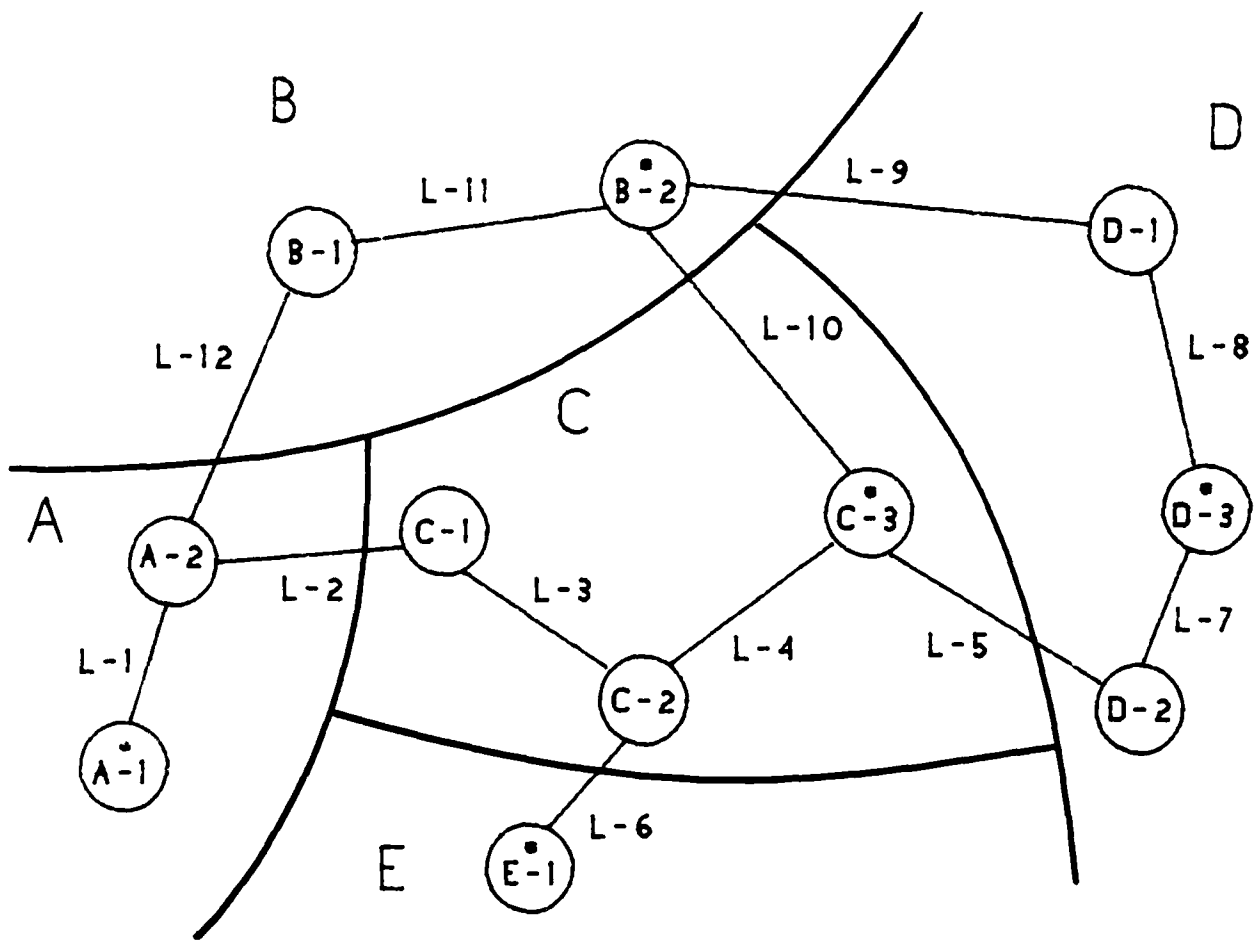


Figure 1: Example Network

designated by a letter-number pair, where the letter indicates the subregion in which the site is located. The communication network links are designated by L-number. The control facility for each subregion is located at the site marked with an "*". Each control facility has a planning agent to restore interrupted service. It should be noted that a separate communication network, of substantially lower bandwidth, is not shown, but is assumed to interconnect the control facilities for the purpose of exchanging messages among the agents.

For the purposes of describing the restoral problem, we assume that there is an equipment malfunction at station B-2 that fails all communication using link L-11. We also assume that each link can handle at most two circuits and that there are four circuits established at the time of the supposed failure. These are described in Table 1 by listing the sites and links along the route of each circuit. To simplify the presentation, these circuits all have the same restoration priority so that none of them should be preferred over the others for restoral in the event of service disruption.

As a result of the presumed failure, two circuits are disrupted, namely ckt-1 and ckt-2 (both use L-11 to get from B-1 to B-2). The planning activity is initiated when an agent observes disruption of a circuit terminating within its subregion and instantiates a restoral

ckt-1	(A-1 :L-1: A-2 :L-12: B-1 :L-11: B-2)
ckt-2	(B-1 :L-11: B-2 :L-10: C-3 :L-5: D-2)
ckt-3	(E-1 :L-6: C-2 :L-4: C-3 :L-5: D-2 :L-7: D-3)
ckt-4	(B-1 :L-12: A-2 :L-2: C-1 :L-3: C-2)

Table 1: Circuit Descriptions

goal. In this example, the restoral goals are autonomously instantiated in subregion A (for ckt-1), subregion B (for ckt-1 and ckt-2) and subregion D (for ckt-2). Each agent initially has only the following knowledge about *each circuit terminating in its subregion*:

- a circuit identifier that is unique within the network,
- a priority or degree of urgency for restoral,
- detailed routing of this circuit within this agent's area of responsibility, and
- the end stations of the circuit and the agents responsible for them.

In addition, each agent has detailed knowledge concerning the status of resources resident in its subregion.

The first phase of the planning process is plan generation, and since it uses only one stage of negotiation, as in contract nets [2,3,4], we shall not consider the details of plan generation here. When viewed from a global perspective, plan generation produces two alternative restoral plans for each circuit. Each plan is represented in Table 2 as a list of alternating sites and links, traversing the proposed restoral path. To clarify the example,

Plans for goal g1 to restore ckt-1:									
g1/p1	(A-1 :L-1: A-2 :L-2: C-1 :L-3: C-2 :L-4: C-3	:L-5: D-2 :L-7: D-3 :L-8: D-1 :L-9: B-2)							
g1/p2	(A-1 :L-1: A-2 :L-2: C-1 :L-3: C-2 :L-4: C-3	:L-10: B-2)							
Plans for goal g2 to restore ckt-2:									
g2/p1	(B-1 :L-12: A-2 :L-2: C-1 :L-3: C-2 :L-4: C-3	:L-10: B-2 :L-9: D-1 :L-8: D-3 :L-7: D-2)							
g2/p2	(B-1 :L-12: A-2 :L-2: C-1 :L-3: C-2 :L-4: C-3	:L-5: D-2)							

Table 2: Alternative Plans

we have adopted a naming convention for goals and alternative plans which incorporates the circuit name and plan number; thus the two alternative plans for restoring circuit ckt-1 are designated g1/p1 and g1/p2. It is essential to remember that these are global plans

which have been generated in a distributed manner, and no single agent *necessarily* knows of all plans or any one complete plan.

As a result of plan generation, a node produces local alternative plan fragments which may be used to satisfy global goals. Each global plan listed in Table 2 is composed of several fragments distributed over a subset of the agents. This is illustrated in Table 3 which summarizes the knowledge each agent has about goals, alternative plan fragments,

Goal	Plan Frag.	Resources Used	Cost
g1	1A	L-1, L-2	9
g2	7A	L-2, L-12	9
Agent A			
g1	2B	L-9	9
	5B	L-10	6
g2	8B	L-9, L-10, L-12	9
	11B	L-12	6
Agent B			
g1	3C	L-2, L-3, L-4, L-5	9
	6C	L-2, L-3, L-4, L-10	6
g2	9C	L-2, L-3, L-4, L-10	9
	12C	L-2, L-3, L-4, L-5	6
Agent C			
g1	4D	L-5, L-7, L-8, L-9	9
g2	10D	L-7, L-8, L-9	9
	13D	L-5	6
Agent D			

Table 3: Local Knowledge About Plan Fragments

and local resources. Plan fragments are numbered and each is identified by a letter indicating the responsible agent. Note that agents are not *explicitly* aware of global alternative plans, but are only aware of local alternatives. For example, even though Agent A has resources needed by both g1/p1 and g1/p2, the local plan fragment is the same in both cases, and thus Agent A "sees" only one alternative plan for goal g1.

This example is considerably oversimplified in order to focus attention on the significant characteristics of this planning problem and to illustrate the cooperation strategy which results from multistage negotiation. The communication network has been simplified so that link capacity is the only resource, and thus there are no constraints arising from local equipment configurations. The number of circuits and link capacities are also much smaller than is typical. Since only two top level goals exist, the subgoal interactions are simple and can be recognized in only one step. In a more realistic problem, subgoal interactions often involve multiple dependencies and may require several steps of negotiation to detect and resolve.

The features of the planning problem which are important for the discussion of multi-

stage negotiation in this paper are summarized below:

- Goals are autonomously generated at nodes in the system.
- The same system goal may be generated at more than one node, independently.
- Knowledge about local resource availability and potential goal interactions at each node differs from that at other nodes.
- Goal satisfaction in general requires nonlocal resources.
- The planning problem being addressed is, in general, overconstrained. A choice to satisfy some goals may preclude the satisfaction of others, so choice heuristics are necessary.
- Goals are prioritized, but this does not imply a total ordering with respect to priority.

4 MODEL OF PROBLEM SOLVING

The planning problem discussed in the previous section can be viewed in a broader context. In this section we characterize a problem solving model in which multistage negotiation is useful. The search space for a problem of this kind can be considered from two points of view: a task or goal centered perspective and a node centered perspective. Each of these ways of viewing the search space provides a different set of insights with respect to problem solving.

When viewed from the perspective of the system goal, the global problem appears as an AND-OR tree progressing from the system goal (at the root), down through goals and plans, to local plan fragments distributed among the agents. A goal centered view of our example problem is illustrated in Figure 2. Two goals have been instantiated, with four alternative plans and several local plan fragments. Of course, since this is a distributed environment, no single agent has a complete view of this tree. Observe that each agent is aware of both goals g_1 and g_2 , but agent D is only aware of one plan fragment for g_1 , the one which is a component of g_1/p_1 .

An agent may not simply satisfy a local goal by choosing any plan fragment, but must coordinate its choice so that it is compatible with those of other agents. Formulation of a plan as a conjunction of plan fragments induces a set of compatibility constraints on the local choices an agent makes in satisfaction of global goals. In Figure 2, we show the plan fragments interconnected by dashed lines. These dashed lines indicate the local knowledge an agent has about which other agents are involved in compatibility constraint relations with its own plan fragments. Observe that an agent generally does not have *complete* knowledge about these compatibility sets. In our application domain, these constraints involved shared resources between two agents.

From a node centered perspective, plan fragment selection is constrained by local resource availability. An agent cannot choose to execute a set of alternative plan fragments that require more local resources than are available. For example, agent B's local resources permit selection of any pair of its own plan fragments in satisfaction of g_1 and g_2 , whereas

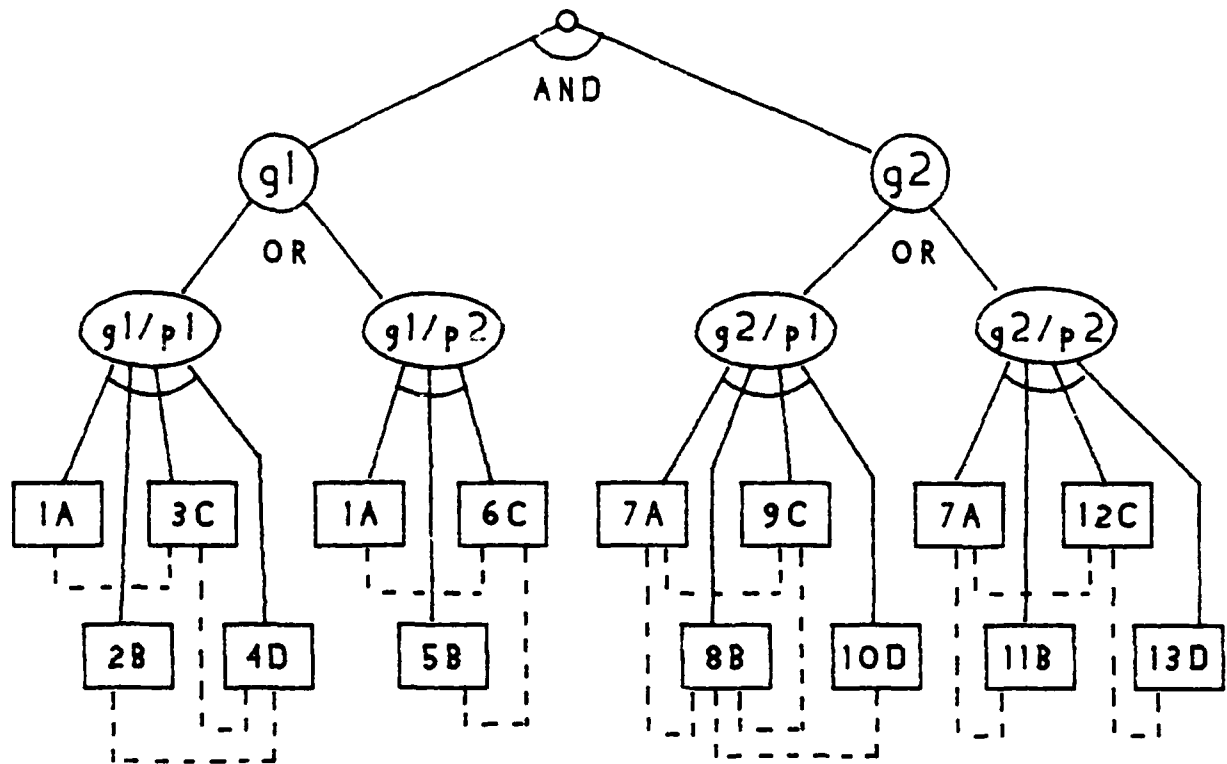


Figure 2: Global Search Space

agents A, C, and D each can select only one plan fragment. The resulting feasibility tree known to each agent is shown in Figure 3. In this figure, resource constraints associated with goals and plan fragments are enclosed by ovals and connected to the appropriate objects with dashed lines. Restoral goals initiated in a subregion are designated with an "**".

From each agent's perspective, the search is over a group of alternatives subject to a set of local resource constraints and a set of compatibility constraints imposed by actions of other agents. Multistage negotiation provides a mechanism by which agents coordinate their actions in selecting plans subject to both resource and compatibility constraints. As additional constraints are added to an agent's base of knowledge, its local feasibility tree is augmented to reflect what it has learned.

5 MULTISTAGE NEGOTIATION

In this section, we describe the multistage negotiation protocol we have developed and give an example of its application in the distributed planning problem which has been discussed. We first treat the protocol at a very high level, discussing the general strategy. We then provide more detail as to phases of planning and the role of negotiation in each. The section is concluded with a detailed trace of negotiation and reasoning in each agent pertinent to our simple example.

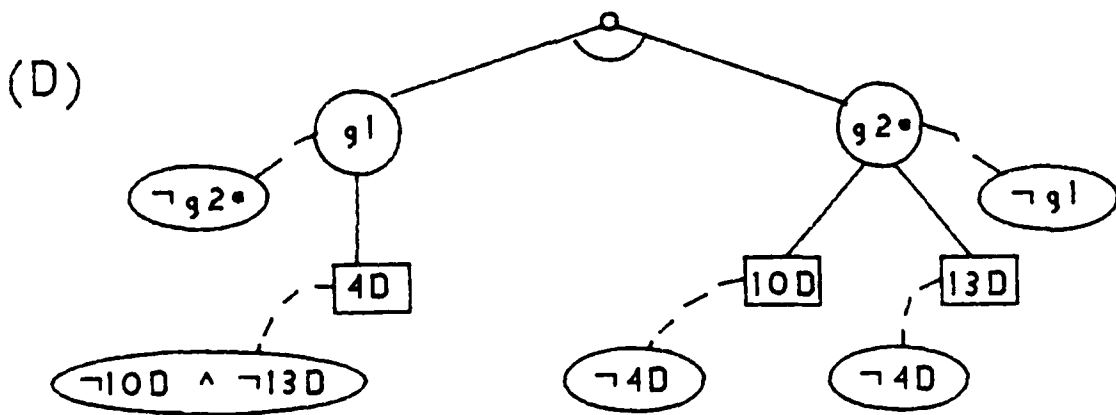
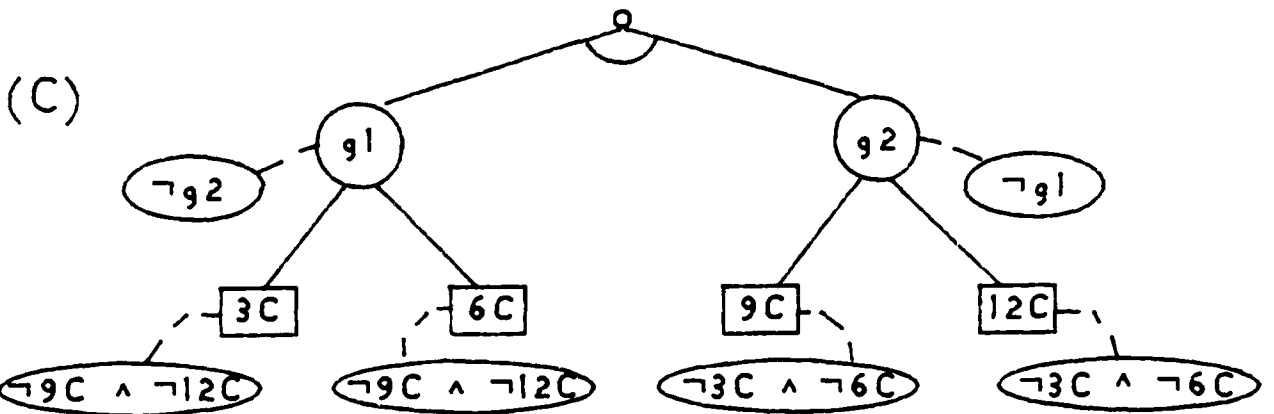
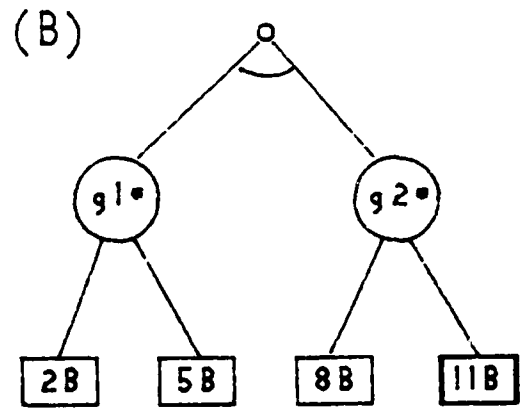
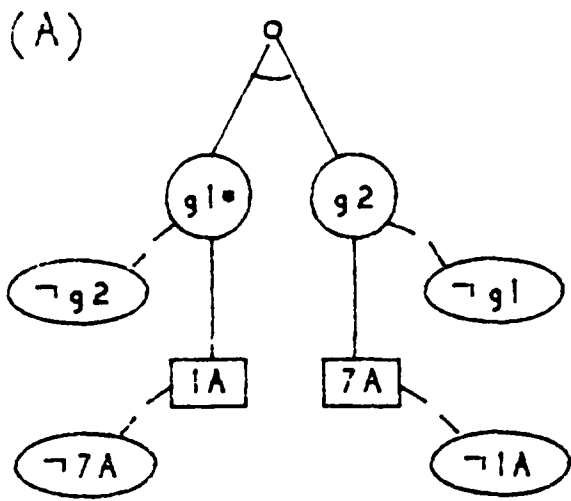


Figure 3: Local Feasibility Trees

High Level Protocol

Multistage negotiation provides a means by which an agent can acquire enough knowledge to reason about the impact of local activity on nonlocal state and modify its behavior accordingly. When problem solving activity is initiated, agents first engage in a phase of plan generation. Each agent ascertains what alternatives for partial goal satisfaction are locally possible and tenders contracts to appropriate agents for furthering satisfaction of the goals needed to complete these plans. On completion of this phase, a space of alternative plans has been constructed which is distributed among the agents, with each agent only having knowledge about its local plan fragments. An agent then examines the goals it instantiated and makes a tentative commitment to the highest rated feasible set of plan fragments relative to these goals. It subsequently issues requests for confirmation of that commitment to agents who hold the contracts for completion of these plan fragments.

Each agent may receive two kinds of communications from other agents: 1) requests for confirmation of other agents' tentative commitments, and 2) responses concerning the impact of its own proposed commitments on others. Impact of local actions is reported as confirmation that a tentative local choice is a good one or as negative information reflecting nonlocal resource conflict. The agent incorporates this new knowledge into its local feasibility tree. It rerates its own local goals using the new knowledge and possibly retracts its tentative resource commitment in order to make a more informed choice. This process of information exchange continues until a consistent set of choices can be confirmed.

Termination of the negotiation process can be done using system-wide criteria or it can be accomplished in a diffuse manner. If global termination criteria are desired in an application, some form of token passing mechanism [8,9,10] can be used to detect that the applicable termination criteria have been met. When synchronized global termination is not required in an application, the negotiation can be terminated by an "irrevocable" commitment of resources. A node initiates plan execution in accordance with its negotiated tentative commitment at some time after it has no pending activities and no work to do for other agents.

Mechanics of Negotiation

When a node begins its planning activity, it has knowledge of a set of top level goals which have been locally instantiated. A space of plans to satisfy each of these goals is formulated during plan generation without regard for any subgoal interaction problems. After plan generation, each node is aware of two kinds of goals: *primary goals* (or p-goals) and *secondary goals* (or s-goals). In our application, p-goals are those instantiated locally by an agent in response to an observed outage of a circuit for which the agent has primary responsibility (because the circuit terminates in the agent's subregion). These are of enhanced importance to this agent because they relate to system goals which *must* be satisfied by this particular agent if they are to be satisfied at all. An agent's s-goals are those which have been instantiated as a result of a contract with some other agent. An agent regards each of its s-goals as a possible alternative to be utilized in satisfaction of some other agent's p-goal.

A plan commitment phase involving multistage negotiation is initiated next. As this

phase begins, each node has knowledge about all of the p-goals and s-goals it has instantiated. Relative to each of its goals, it knows a number of alternatives for goal satisfaction. An alternative is comprised of a local plan fragment, points of interaction with other agents (relative to that plan fragment), and a measure of the cost of the alternative (to be used in making heuristic decisions). Negotiation leading to a commitment proceeds along the following lines.

1. Each node examines its own p-goals, making a tentative commitment to the highest rated set of locally feasible plan fragments for p-goals (s-goals are not considered at this point because some other agent has corresponding p-goals).
2. Each node requests that other agents attempt to confirm a plan choice consistent with its commitment. Note that an agent need only communicate with agents who can provide input relevant to this tentative commitment.
3. A node examines its incoming message queue for communications from other nodes. Requests for confirmation of other agents' tentative commitments are handled by adding the relevant s-goals to a set of active goals. Responses to this agent's own requests are incorporated in the local feasibility tree and used as additional knowledge in making revisions to its tentative commitment.
4. The set of active goals consists of all the local p-goals together with those s-goals that have been added (in step 3). The agent rates the alternatives associated with active goals based on their cost, any confirming evidence that the alternative is a good choice, any negative evidence in the form of nonlocal conflict information, and the importance of the goal (p-goal, s-goal, etc.). A revised tentative commitment is made to a highest rated set of locally consistent alternatives for active goals. In general, this may involve decisions to *add* plan fragments to the tentative commitment and to *delete* plan fragments from the old tentative commitment. Messages reflecting any changes in the tentative commitment and perceived conflicts with that commitment are transmitted to the appropriate agents.
5. The incoming message queue is examined again and activity proceeds as described above (from step 3). The process of aggregating knowledge about nonlocal conflicts continues until a node is aware of all conflicts in which its plan fragments are a contributing factor.

Two issues need clarification at this point. One deals with the question of termination and the other is concerned with the quality of the result obtained through negotiation (relative to optimality).

Negotiation in this framework continues as long as there are any pending activities in an agent. The only way a situation leading to nontermination could arise involves an agent's making a tentative commitment and subsequently entering a cycle of retracting and remaking that commitment indefinitely. It is not reasonable to expect that an agent should never retract a tentative commitment. It is also not reasonable to expect that an agent would never decide, based on new knowledge, to recommit to an alternative it had previously rejected. An agent's local reasoning must be able to detect when it is making a tentative

commitment it has previously made with no new knowledge. Negotiation activity in an agent terminates either when it has no pending activity and no incoming communications or if an attempt is made to return to a previous commitment with no new knowledge from other agents. Endless loops of commitment and decommitment are prevented through this mechanism.

The other issue of importance at this point is related to the quality of the result obtained through negotiation. In the initial negotiation stage, each agent examines only its p-goals and makes a tentative commitment to a locally feasible set of plan fragments in partial satisfaction of those goals. Since each agent is considering just its p-goals at this stage, the only reason for an agent's electing not to attempt satisfaction of some top level goal is that two or more of these goals are locally known to be infeasible. (This corresponds to an overconstrained problem.)

In subsequent stages of negotiation, both p-goals and relevant s-goals are considered in making new tentative commitments. The reasoning strategy employed at each agent will only decide to forego commitment to one of its p-goals if it has learned that satisfaction of this p-goal precludes the satisfaction of one or more other p-goals elsewhere in the system. *If the system goal of satisfying all of the p-goals instantiated by agents in the network is feasible, no agent will ever be forced to forego satisfaction of one of its p-goals (because no agent will ever learn that its p-goal precludes others), and a desired solution will be found.* If, on the other hand, the problem is overconstrained, some set of p-goals cannot be satisfied and the system tries to satisfy as many as it can. While there is no guarantee of optimality, the heuristics employed should ensure that a reasonably thorough search is made.

To make these concepts concrete, multistage negotiation is applied to the simplified planning problem discussed in the previous sections.

Example

We return to our example of planning activity, assuming that each agent has the knowledge depicted in the appropriate part of Figure 3. A summary of the transactions that occur during negotiation to achieve plan selection is shown in Table 4. This table is segmented by agent and by "time slice" to convey a sense of progress in problem solving through negotiation. The notational conventions are relatively simple. Tentative commitment to a locally known activity and the associated communication issued to an appropriate agent is denoted in the form (plan fragment name; message → agent). Exchange of conflict information is indicated in the form (conflict; type of conflict → agent). To make the trace easier to follow, each received message is noted in the form (source agent → message). As is evident in Table 4, negotiation begins with tentative commitments to alternatives in agents A, B, and D. Though the problem is overconstrained (it is not possible to restore both ckt-1 and ckt-2), no agent is yet aware of that fact. In response to the initial tentative commitments, there is activity in agents A and C. Agent A knows that it cannot act to satisfy both g1 and g2, but it does *not* know if this precludes satisfaction of g2 (since g2 is an s-goal, there might exist another global plan not requiring any action by A). Since A recognizes the need to *attempt* satisfaction of its own p-goal first, agent A informs agent B there is a conflict between what B requested and satisfaction of one of A's p-goals. Thus A has given B the knowledge that the plan fragment B selected would force A to forego one

A	B	C	D
1A; OK? → C	11B; OK? → A 5B; OK? → C		13D; OK? → C
B → OK? 11B conflict; (11B AND ¬ p-goal g1) → B		A → OK? 1A B → OK? 5B D → OK? 13D match 6C with 1A and 5B 1A is OK → A 5B is OK → B conflict; (13D AND ¬ g1 via C) → D	
C → 1A is OK	A → (11B AND ¬ p-goal g1) C → 5B is OK 8B; OK? → A		C → (13D AND ¬ g1 via C) 10D; OK? → B
B → OK? 8B conflict; (8B AND ¬ p-goal g1) → B	D → OK? 10D 8B; OK? → C		
	A → (8B AND ¬ p-goal g1) B knows g1 and g2 not both possible (not both g1 and g2) → D	B → OK? 8B conflict; (8B AND ¬ g1 via C) → B	
	C → (8B AND ¬ g1 via C)		B → (not both g1 and g2)

Table 4: Summary of transactions during negotiation

of its p-goals.

Agent C has now received three communications requesting that plan fragments be extended. It observes that it can effect a plan completion for g1, satisfying both the request from A and the request from B. It also observes that it cannot satisfy both g1 and g2 with use of its locally known plan fragments due to local resource constraints. Since it has the opportunity to complete a plan for ckt-1 and not for ckt-2, it elects to tentatively commit its resources to plan fragment 6C. Messages reflecting this commitment are formulated and transmitted to A and B, while a message indicating the conflict in C is sent to D.

As a result of this second round of communications, activity in subregions B and D is concerned with exploring the remaining alternatives they have for restoral of ckt-2. An acceptable plan for ckt-1 is already reflected in tentative commitments. Agent B elects to try plan fragment 8B and agent D elects to try 10D. Agent B learns that an attempt to satisfy g2 via 8B also fails in A, so it now knows that the problem is overconstrained. Based on the fact that a way of satisfying g1 has already been located, B elects to forego satisfaction of g2 and advises D that it should also give up on g2. Negotiation terminates with tentative commitments reflecting a plan choice for g1.

In concluding this section we summarize, by "time slice", changes to the local feasibility trees that take place during the negotiation illustrated in Table 4.

Slice 1:

- No changes.

Slice 2:

- No changes in constraints by A.
- 6C is tentatively committed to a *complete* plan by C.

Slice 3:

- 1A is marked as tentatively satisfying g1 by A.
- 5B is marked as tentatively satisfying g1 by B.
- Agent B adds the constraint (\neg g1) to 11B.
- Agent D adds the constraint (\neg g1 via C) to 13D. (Note that in this example, the new constraint on 13D is, in fact, redundant. In other examples, with a more complex set of goals, new constraints propagated in this way often provide additional information.)

Slice 4:

- No changes.

Slice 5:

- Agent B adds the constraint (\neg g1) to 8B.
- Agent B propagates the constraint (\neg g1) on 8B and 11B to their parent, g2. *Agent B now knows the problem is overconstrained.*

Slice 6:

- Agent D modifies the constraint ($\neg g_1$) on goal g_2^* to ($\neg g_1^*$). *Agent D now knows the problem is overconstrained.*

This example illustrates ways in which knowledge is integrated into the local feasibility tree as it is acquired through negotiation. It shows how knowledge aggregated at the level of plan segments can be propagated in drawing inferences concerning interactions at the goal level. It also shows how the network of agents can become aware that it has an overconstrained problem.

6 CONCLUDING REMARKS

In this paper, we have presented a new paradigm for cooperation in distributed problem solving systems. This paradigm incorporates features found in two cooperation strategies treated in the literature: the contract net protocol [2,3,4] and the FA/C paradigm [5]. It has been devised to permit an agent in a distributed problem solving system to acquire enough knowledge to reason about the impact of local activity on nonlocal state and to modify its behavior accordingly.

Three characteristics of distributed planning problems motivate development of a more general cooperation paradigm. First, subgoal interaction problems that arise in the context of a distributed planning system in which agents do not have a global view are very difficult to detect and even more difficult to handle in a reasonable way. Second, many application domains embody planning problems that are overconstrained. When these planning problems are addressed by a network of planning agents, it is essential that the system be able to determine whether or not the problem is overconstrained. Third, when the planning problem is overconstrained, it is necessary for the agents involved to arrive at an agreement as to a set of goals whose satisfaction is regarded as an acceptable solution to the problem at hand. None of these issues can be resolved in the context of the previously proposed cooperation paradigms without the exchange of sufficient knowledge as to permit each agent to construct a global view.

Another factor motivating formulation of a more general cooperation paradigm is the observation that many application domains have characteristics that distinguish them from other multi-agent planning problems which have been investigated. The strategies suggested by Lansky [11] and Georgoff [12] dealing with planning for a multiple agent domain by a centralized planner are not applicable in situations where there is no central planner. In addition, the agents in our networks are not motivated purely by self interest. They are interested in cooperating to achieve some goals pertinent to system performance. For this reason, the metaphor proposed by Genesereth and others [13] does not represent the domain characteristics. It should be noted, however, that our metaphor can be adapted for use in networks of agents which are selfish (as long as they do not lie a great deal).

The mechanisms presented in this paper are related to the techniques that have been utilized in conventional planning systems. Each agent in our system builds a data structure analogous to the Table of Multiple Effects used by NOAH [14] and NONLIN [15] in detecting subgoal interactions. This structure is incrementally built using knowledge gleaned through

negotiation. In detecting and resolving conflicts, a form of criticism analogous to that performed by NOAH's Resolve Conflicts critic is employed. Criticism is necessary in our distributed problem solving systems for the same reason it was needed in NOAH - decisions are made initially based on local criteria, whereas nonlocal conditions affect the viability of those decisions. Unlike NOAH (and like NONLIN), alternatives are not discarded after they have been rejected. Backtracking in the form of revised tentative commitment is a feature of the protocol.

In many planning problems, the constraints arising from resource availability are very important in determining a satisfactory solution to the planning problem. We have found that resource constraints play a crucial role in our system as well. The ability to reason about resources is critical in determining adequate solutions. This was recognized in the design of SIPE [16]. Since we have no central planner, the mechanisms for reasoning about resources are somewhat different from those employed in SIPE, but resources as a factor in problem solving are just as important to multistage negotiation as they were in SIPE.

The distributed planning system discussed in this paper is currently in the final stages of implementation on an existing distributed system simulation facility [17].

References

- [1] J. G. March and H. A. Simon, **Organizations**, Wiley, 1958.
- [2] Reid G. Smith "The Contract Net Protocol: High Level Communication and Control in a Distributed Problem Solver," **IEEE Transactions on Systems, Man, and Cybernetics**, vol. SMC-10, no. 12, December 1980.
- [3] R. G. Smith and R. Davis "Frameworks for Cooperation in Distributed Problem Solving," **IEEE Transactions on Systems, Man, and Cybernetics**, vol. SMC-11, no. 1, January 1981, pp. 61-70.
- [4] R. Davis, and R. G. Smith "Negotiation as a Metaphor for Distributed Problem Solving," **Artificial Intelligence**, vol. 20, no. 1, January 1983, pp. 63-109.
- [5] V. R. Lesser and D. D. Corkill "Functionally Accurate, Cooperative Distributed Systems," **IEEE Transactions on Systems, Man, and Cybernetics**, vol. SMC-11, no. 1, January 1981, pp. 81-96.
- [6] S. E. Conry, R. A. Meyer, and J. E. Searleman "A Shared Knowledge Base for Independent Problem Solving Agents," **Proceedings of the Expert Systems in Government Symposium**, IEEE Computer Society, McLean, Virginia, October 1985.
- [7] A. S. Tanenbaum, **Computer Networks**, Prentice-Hall, 1981.
- [8] S. Vinter, K. Ramamritham, and D. Stemple "Recoverable Communicating Actions in Gutemberg," **Proceedings of the International Conference on Distributed Computing Systems**, May 1986.
- [9] E. W. Dijkstra and C. S. Scholten "Termination Detection for Diffusing Computations," **Information Processing Letters**, vol. 11, no. 1, August 1980, pp. 1-4.

- [10] N. Francez "Distributed Termination," **ACM Transactions on Programming Languages and Systems**, vol. 2, no. 1, January 1980, pp. 42-55.
- [11] A. L. Lansky "Behavioral Specification and Planning for Multiagent Domains," Technical Note 360, Artificial Intelligence Center, SRI International, Menlo Park, California, November 1985.
- [12] M. Georgeoff, "Communication and Interaction in Multi-Agent Planning," **Proceedings of the National Conference on Artificial Intelligence (AAAI-84)**, August 1984, pp. 125-129.
- [13] M. R. Genesereth, M. L. Ginsberg, and J. S. Rosenschein "Cooperation Without Communication," **Proceedings of the National Conference on Artificial Intelligence (AAAI-86)**, August 1986, pp. 51-57.
- [14] E. D. Sacerdoti **A Structure for Plans and Behavior**, Elsevier-North Holland, New York, 1977.
- [15] A. Tate "Project Plannig using a Hierarchic Non-Linear Planner," Research Report No. 25, Department of Artificial Intelligence, University of Edinburgh, August 1976.
- [16] D. E. Wilkins, "Domain-Independent Planning: Representation and Plan Generation," **Artificial Intelligence**, vol 22, pp. 269-301.
- [17] D. J. MacIntosh and S. E. Conry, "SIMULACT: A Generic Tool for Simulating Distributed Systems" to appear in the **Proceedings of the Eastern Simulation Conference**, April 1987.

Knowledge Acquisition as Knowledge Assimilation

Lawrence S. Lefkowitz
Victor R. Lesser

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

`larry@umass-cs.edu`

`lesser@umass-cs.edu`

April 1987

Abstract

Knowledge bases are constructed and refined using information obtained from domain experts. The assimilation of this information into an existing knowledge base is an important facet of the knowledge acquisition process. Knowledge assimilation requires an understanding of how the new information corresponds to that already known by the system and how this existing information must be modified so as to reflect the expert's view of the domain.

This paper describes a system, K^{NAIC} , that modifies an existing knowledge base based on a discourse with a domain expert. Using heuristic knowledge about the knowledge acquisition process, K^{NAIC} anticipates modifications to the existing entity descriptions. These anticipated modifications, or *expectations*, are used to provide a context in which to assimilate the new domain information.

This research was supported, in part, by the External Research Program of Digital Equipment Corporation, by the Air Force Systems Command, Rome Air Development Center, Griffiss Air Force Base, New York 13441-5700, and by the Air Force Office of Scientific Research, Bolling AFB, DC 20332 under Contract No. F 30602-85-C-0008. This contract supports the Northeast Artificial Intelligence Consortium (NAIC).

1 Introduction

An often overlooked aspect of the knowledge acquisition process is the assimilation of information presented by the domain expert into an existing knowledge base. Consider the typical means by which knowledge bases are currently constructed. This usually involves a series of dialogs between an expert, or experts, in the application domain and a knowledge engineer familiar with the target expert system. The knowledge engineer's task is the modification of the expert system's knowledge base so as to reflect the domain expert's knowledge. To a large extent, this knowledge acquisition task may be viewed as a recognition problem. All of the problems facing other recognition systems are present here as well, including: noisy data (i.e., incomplete or inaccurate information), ambiguous interpretations, and the need to produce intermediate results before all the data is available. Thus, a significant portion of this interactive knowledge acquisition task is a matching problem: How does the expert's description of the domain correlate with the description contained in the knowledge base? How should the knowledge base be modified based on new information from the expert? What should be done when the expert's description differs from the existing one?

K^{N}_{Ac} is a system that we have developed that implements this knowledge assimilation approach to knowledge acquisition. It was developed to assist in the construction of knowledge bases for the POISE [CLLH82] intelligent interface system. These knowledge bases use a frame-like representation, described more fully in [CL84,Lef87] to describe *tasks*, *objects* and *relationships* in the application domain. POISE's initial knowledge bases, for the office automation and software engineering domains, were created by hand from interviews between a knowledge engineer and the appropriate domain experts. Transcriptions of these interviews were examined and the results served as the basis of the K^{N}_{Ac} system.

It is important to note that the goal of the domain expert was *not* to modify POISE's knowledge base; this was the knowledge engineer's role. The expert simply presented the domain information, e.g., descriptions of tasks, objects, etc., and responded to questions

<p>EVENT TAKE-A-TRIP-AND-GET-PAID</p> <p><i>STEPS: (TAKE-A-TRIP GET-REIMBURSED)</i></p> <p><i>TEMPORAL-RELATIONSHIPS:</i></p> <p><i>((TAKE-A-TRIP before GET-REIMBURSED))</i></p> <p><i>CONSTRAINTS: (...)</i></p> <p><i>ATTRIBUTES: ((TRAVELER ...) (COST ...) (DESTINATION ...))</i></p>

Figure 1: Knowledge Base Event Description

and comments from the knowledge engineer. The burden of assimilating the information, that is, recognizing where it fit into the existing knowledge base and what additions or modifications were needed, was not placed upon the domain expert. (Contrast this to approaches such as [EEMT86,GWP85,KNM85].) By modeling the knowledge engineer's role in this task, Kⁿ_Ac attempts to provide this same support.

Consider the opening portion of a discourse in which the expert, the principal clerk of an academic department, is describing the procedure for being reimbursed for business-related travel expenses.

"O.K. — on travel. The proper way of doing it, if it's out of state, is that a travel authorization should be issued before the trip."

From this information one can conclude that some unnamed task consists of two temporally ordered steps. However, it is not clear what modifications need be made to the knowledge base to reflect this information.

If the knowledge base is examined (prior to this interview), a description of this reimbursement process will be found (see Figure 1). In this simplified view of the task, which knows nothing about a "travel authorization", the traveler simply goes on a trip and gets reimbursed. Though the knowledge engineer may realize that the clerk and this description are describing the same task, it is not readily apparent from the two descriptions. Matching such descriptions, and recognizing the implied modifications, are

central to the assimilation process.

To accomplish this, K^{nAc} was required to perform two basic tasks: 1) recognizing where the expert's information fits into the existing knowledge base, and 2) appropriately modifying the existing knowledge so that it reflects the expert's view of the domain. Determining where the expert's information fits into the existing knowledge requires that the new information be matched against the existing information. To avoid matching the new information against the entire (existing) knowledge base, the most likely candidate matches must be selected. Furthermore, since the goal of a knowledge acquisition discourse is the modification of the knowledge base, exact matches between the new and the existing information are not always expected.

Thus, the procedure for matching the expert's entity descriptions with those already in the knowledge base must be specialized for knowledge acquisition. K^{nAc} 's matching and match evaluation procedures are described in Section 3. Discrepancies between these descriptions may imply needed modifications and need not degrade a match, especially if the discrepancies (or the implied modifications) can be predicted. Anticipated modifications, or *expectations*, arise from an understanding of the knowledge acquisition process. They can be derived from the state of the existing knowledge base, from cues in the discourse, from previous modifications to entity descriptions, or from the state of the knowledge acquisition task. The generation and management of these expectations is described in Section 4. Finally, the status of this work and its contributions to the knowledge acquisition task are summarized in Section 5.

2 The K^{nAc} System

In this section, the basic architecture and functionality of the K^{nAc} system¹ is presented. During each cycle of the K^{nAc} system, descriptions of domain entities are accepted from the

¹Figure 2 contains the architecture of the knac system. The parenthesized numbers in this paragraph (e.g., (1)) refer to this figure.

user (1) and compared with entities in the existing knowledge base (2). (Figure 4 contains a portion of this knowledge base.) These candidate entities (3) are selected based on K^{nAc} 's expectations of changes to the knowledge base. The comparisons (4) are evaluated both in terms of how well they match and the extent to which the differences between them were expected (5) within the context of the match. Once the best matches are selected, the implied modifications (6) are made to the existing entity knowledge base (7), after being verified with the user (8), if necessary. Expectations of further modifications are generated from a variety of sources, including the information obtained from the discourse (9), the state of entities in the knowledge base (10), previously made modifications (11) and the state of the acquisition process.

The descriptions obtained from the expert must be presented to the matcher in the knowledge base's representation language. The purpose of the *discourse manager / user interface* is twofold: to permit a more "user-friendly" specification (e.g., natural language, graphics, menus, etc.) of these descriptions, and to provide K^{nAc} with any available cues as to the state of the discourse. Currently, the natural language protocols are translated, by hand, into the system's representation language. Discourse cues are assumed to be minimal, such as "topic" information.

Thus, the discourse fragment presented in Section 1 translates, approximately, into the structures shown in Figure 3. These structures may then be compared with selected entities from the existing knowledge base.

To avoid having to examine the entire knowledge base in order to assimilate the new information, entities that are most likely to be modified are selected as candidate matches. Thus, if there exists an expectation of some modification to a given entity description, that entity is compared to the new information from the expert. The way in which these modifications are anticipated will become clearer in Section 4. Initially, the only expectations available are based on the discourse cue recognizing that TRAVEL is a topic of interest. Hence, the entities semantically close to TRAVEL in the knowledge base are selected as candidate matches. These entities include TAKE-A-TRIP-AND-GET-PAID,

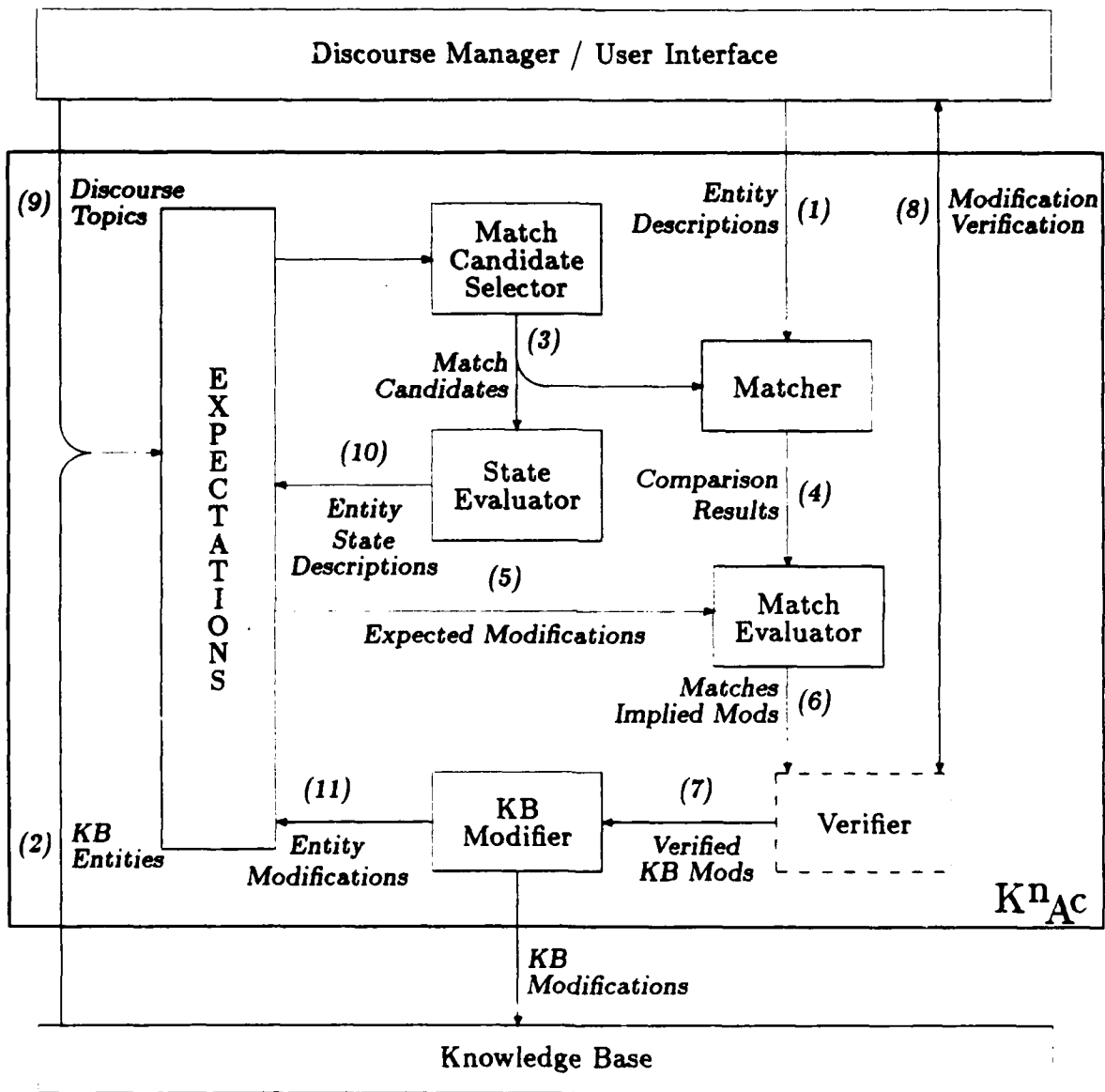


Figure 2: The KⁿAc System Architecture

EVENT EVENT-1**STEPS:** (ISSUE-TRAVEL-AUTHORIZATION TAKE-A-TRIP)**TEMPORAL-RELATIONSHIPS:**((ISSUE-TRAVEL-AUTHORIZATION *before* TAKE-A-TRIP))**CONSTRAINTS:** ((DESTINATION *outside-of* STATE))**ATTRIBUTES:** ((TRAVELER ...) (DESTINATION ...))**EVENT ISSUE-TRAVEL-AUTHORIZATION****EVENT TAKE-A-TRIP****OBJECT TRAVEL-AUTHORIZATION**

Figure 3: Discourse Manager Output

shown in Figure 1.

The system then compares the expert's descriptions with the selected match candidates. The matching process, described more fully in Section 3, determines the similarities and differences between a pair of entity descriptions. The results of these comparisons are then evaluated in order to select the best match for each of the expert's entity descriptions. Section 3.2 describes the evaluation process, which rates the match results on a field-by-field basis and combines these ratings to produce an overall rating for each match.

For example, when the task described by the expert, i.e., EVENT-1, is compared with the existing task description TAKE-A-TRIP-AND-GET-PAID, the contents of each field of these structures (e.g., *parts*, *generalizations*, *temporal-relationships*, *pre-* and *post-conditions* etc.) are compared. In the *steps* field, they have one entity in common (TAKE-A-TRIP) and each has one entity not found in the other (ISSUE-TRAVEL-AUTHORIZATION in EVENT-1 and GET-REIMBURSED in TAKE-A-TRIP-AND-GET-PAID). They have several *attributes* in common (TRAVELER and DESTINATION). Their *temporal-relationships* are mutually consistent, though different. Both entities are *specializations* of EVENT.

The ratings for these field matches is shown in Figure 5. Remember, these ratings reflect not only the degree to which the fields match, but more importantly (from the point

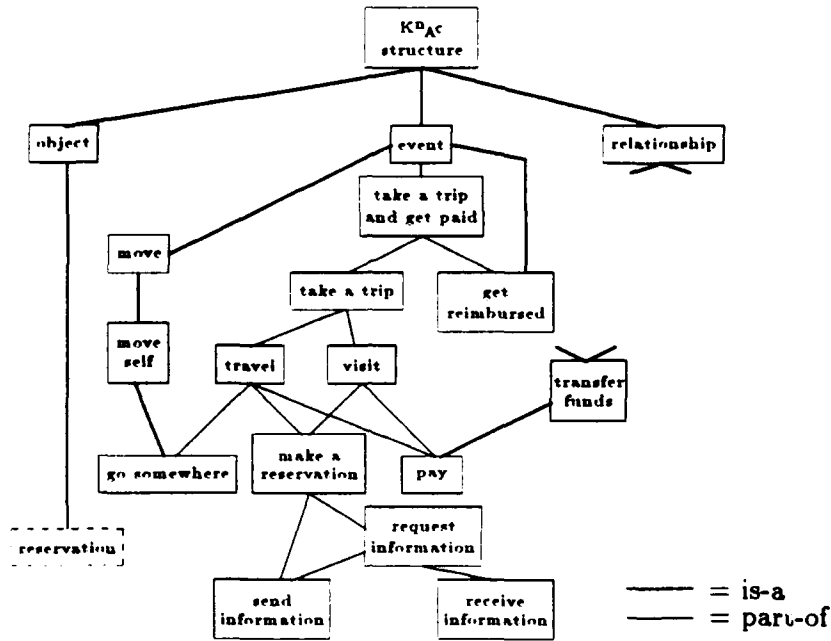


Figure 4: A portion of the knowledge base

of view of knowledge acquisition) the likelihood of the modifications, in future parts of the dialog, required to make them match.

From these ratings, the best matches are selected. If there is significant ambiguity, the matches are verified with the expert. If there are no acceptably good matches for one of the expert's entity descriptions, a new entity is added to the knowledge base.

When the best matches have been selected, the differences between the expert's de-

EVENT-1 vs. TAKE-A-TRIP-AND-GET-PAID

Field	Rating
generalizations	1.000
parts	0.133
attribute-names	0.055
constraints	0.007
Match Rating	0.299

Figure 5: Ratings for field matcher

scription and the existing one are used to modify the knowledge base. For instance, the "extra" step in EVENT-1, i.e., ISSUE-TRAVEL-AUTHORIZATION, is added as a step of TAKE-A-TRIP-AND-GET-PAID. The extent to which this modification requires confirmation from the expert depends on the level of autonomy granted to the system. At various levels, all such modifications could be verified with the expert or only unexpected ones or only deletions, etc.

Once the knowledge base has been modified, new expectations are generated to be used in interpreting the next "discourse frame". The generation and management of these expectations is described in Section 4.

3 Matching for Knowledge Acquisition

In order to match entity descriptions provided by the domain expert to those already known to the system, K_{AC}^n must be able to compare these structures and evaluate the results. This matching process, while in some ways similar to that found in most recognition/interpretation systems, displays certain characteristics unique to knowledge acquisition. In particular, since the goal of a knowledge acquisition dialog is the modification of the knowledge base, the information provided by the domain expert should not completely match the existing entity descriptions. The matching process must be modified so as to be able to recognize and, where possible, anticipate these discrepancies. The matching techniques presented in this section make these discrepancies explicit; the evaluation of these match results, described in Section 3.2, incorporates the extent to which these discrepancies were anticipated into its rating procedure.

3.1 Matching Entity Descriptions

POISE's knowledge base is represented in a frame-based language, similar to that used by systems like Knowledge Craft®[WF83,KC86]. Comparison of these knowledge structures requires matching on a field-by-field basis. Each field may be considered to be one of

two types of structures: *sets of elements* such as *steps* and *generalizations* of an event, or *collections of constraints* such as the *temporal relationships* and the *pre- and post-conditions*. K_{AC}^n contains matching techniques for both of these types of structures.

3.1.1 Set Matching

Determining how well two sets of elements match is not difficult; neither is determining how "different" they are (e.g., see [Tve77]). For knowledge acquisition purposes, however, the relevant question is "How likely is it that they can be modified so as to match?" To determine this, K_{AC}^n examines not only the elements in each (set) field of the knowledge structure, but also makes use of information *about* that field (i.e., *meta-information* or *facets*). In particular, information about the size of each field and the range of the elements in that field permit K_{AC}^n to calculate the probability that the *extra* elements in one of the structures will be added to the other.

Consider the comparison of the *steps* of EVENT-1 and TAKE-A-TRIP-AND-GET-PAID. A typical measure of similarity is:

$$\text{match-rating} = \begin{cases} 1 & \text{if } \text{Set}_1 = \text{Set}_2 = \emptyset \\ \frac{|\text{Set}_1 \cap \text{Set}_2|}{|\text{Set}_1 \cup \text{Set}_2|} & \text{otherwise.} \end{cases}$$

With one step out of the three unique steps between them in common, the similarity of these fields would be 1/3. How likely is it, however, that the "extra" step in the expert's description (i.e., ISSUE-TRAVEL-AUTHORIZATION) will be added to the existing description? Without requiring a deep understanding of domain-specific semantics, some additional information can still be used. If events usually have few steps (as specified by the meta-information about the *step* field of EVENT), the addition of this particular step is less likely than if there are many more steps still to be added. Similarly, if a step is going to be added, the range of possible steps will affect the probability of the desired one being added. This range is determined by combining the "type restriction" meta-information about the slot (i.e., a *step* of an EVENT must be an EVENT) with the existing knowledge

base (i.e., the number of EVENT descriptions known to the system).

Thus, K_{AC}^n rates each set match based on the likelihood of the modifications implied by the match. The likelihood is determined from the anticipated size and range of the sets and requires no additional semantic information about their content. The derivation of these ratings is fully described in [Lef87].

3.1.2 Constraint Matching

Although comparing (or combining) arbitrarily constrained sets of entities is a difficult problem often requiring substantial domain knowledge, K_{AC}^n compares sets of constraints, pairwise related by a common relation, in a purely mechanical fashion.² This section describes a mechanism for comparing such constrained sets; the mechanism is independent of the particular relation, requiring only a description of its algebraic properties, i.e., whether or not the relation is *reflexive*, *symmetric* and/or *transitive*.

First, any implicit constraints are made explicit by propagating the specified constraints using the relation's algebraic properties. The temporal relation *before*, for example, is only transitive; if the constraints (A *before* B) and (B *before* C) were specified, then (A *before* C) could be deduced. When the constraints are thus propagated, inconsistencies may be detected by check the results for any of the prohibited properties of the relation (i.e., *non-reflexive*, *non-symmetric* and *non-transitive*). If each set of constraints is internally consistent, the two sets may be merged and re-propagated, and this combined set of constraints may be checked for consistency.

If two sets of constraints are mutually consistent, a measure of their similarity may be obtained by determining the changes required to make them equivalent. Simply adding each set of constraints to the other would accomplish this, but this may add more infor-

²The current system checks each relation separately; it does not handle interaction between different relations, though it is able to combine relations with their inverses. For instance, *before* and *after* constraints are handled together.

mation than is necessary. For instance, if the first set of constraints contained (*A before B*) and (*A before C*) and the second sets contained (*B before C*), then only (*A before B*) need be added to the second set. Requiring the addition of both constraints from the first set would imply a larger discrepancy between the sets than actually exists. Obtaining the minimal set of constraints that need to be added is not a trivial problem. K_{AC}^n contains a new approach to generating these sets, called "opensures"³, based on their algebraic properties.

Thus, as with fields containing *sets*, K_{AC}^n is able to rate *constraint* matches by determining the likelihood of the implied modifications. This rating is based on the algebraic properties of the relationships involved and requires no additional semantic information. K_{AC}^n 's methods for constraint propagation, determination of consistency, and generation of opensures are presented in [Lef87].

3.2 Match Evaluation

After comparing each of the entity descriptions provided by the domain expert against those candidate entities selected from the existing knowledge base, K_{AC}^n evaluates these match results in two passes. First, the likelihood of two entities matching, based on the extent to which they differ and the probability of these differences being corrected, is determined as described above. This "degree of fit" is a relatively inexpensive means of pruning the set of possible matches.

The second pass of the match evaluation takes into account the context in which the comparison is being made. In addition to *how* the descriptions differ, it considers whether these differences are expected in a particular situation. The extent to which the modifications implied by the differences between the structures are expected serves as a "context-dependent" measure of the match. The anticipation of such modifications is a crucial part of the K_{AC}^n system and is described in the following section.

³i.e., the inverse of "closures"

Consider, for example, the addition of the step ISSUE-TRAVEL-AUTHORIZATION to the description of TAKE-A-TRIP-AND-GET-PAID. The addition of such a step could have been foreseen for several reasons. First, since there were fewer steps in the existing description than are typically found in events, adding another step was reasonable. More importantly, upon examining the existing description to see if it was consistent and complete, it was discovered that a *precondition* of the step GET-REIMBURSED, describing the traveler as the recipient of funds, could not be satisfied by the only earlier step in the task (i.e., TAKE-A-TRIP). This further supported the addition of another step (occurring before GET-REIMBURSED) to the task.

4 Anticipating Modifications

K^{nAc} provides a context in which to interpret information provided by a domain expert by anticipating modifications to an existing knowledge base. These anticipated modifications, or *expectations*, are derived from K^{nAc} 's heuristic information about the knowledge acquisition process. This section describes how these expectations are generated, how they are used to provide a context in which matches may be evaluated, and how they ranked and managed.

4.1 Generating Expectations

K^{nAc} contains a body of heuristics about the knowledge acquisition process. These heuristics, obtained through the analysis of several knowledge acquisition dialogs, allow K^{nAc} to anticipate modifications to an existing knowledge base. These heuristics may be divided into four categories. The first group is based on the state of the knowledge, both that already contained in the knowledge base and new information provided by the expert. The second category depends on modifications previously made to the existing knowledge base. The third set makes use of a model of the discourse process, while the final set incorporates knowledge about teaching and learning strategies.

Since it is expected that the collection of heuristics will be modified, both as a result of improved understanding of the knowledge acquisition process and through the addition of domain specific heuristics, K^{nAc} allows heuristics to be added (or removed) in a straightforward way. Most of K^{nAc} 's current heuristics are quite simple and domain-independent; the addition of more complex, application-specific heuristics may improve the system's performance in certain domains.

Consider a simple heuristic based on the state of an entity description:

Heuristic S2: *Fields with too few components will be augmented.*

This heuristic states that if information is detected to be missing, the addition of that information may be expected. Missing information may be detected in various ways. One simple approach compares the number of entries in a field of a knowledge structure with the field's expected cardinality. If the field is determined to contain too few values, additional values (of the appropriate type) will be expected. The expected field size may come, in order of specificity, from meta-information about a given field of a given entity, via inheritance from a generalization of the entity, from the default information for the type of the entity, or from an overall field default size. This size information may be static or determined dynamically by the system. An expectation generated by this heuristic is:

Exp146: Expecting (certainty 0.360):

MOD: ADD ?New-part<is-a-knac-structure-p> to the
Parts field of Take_a_trip_and_get_paid
Derived from Take_a_trip_and_get_paid and H_S2.

Other heuristics based on the state of the knowledge exploit references to unknown entities, unsatisfied preconditions, and range/value conflicts to anticipate changes.

Changes to the knowledge base may imply additional modifications. For instance, two heuristics that are triggered when a new entity description is added are:

Heuristic M1: *Detailed information usually follows the introduction of a new entity.*

Heuristic M2: *Context information usually follows the introduction of a new entity.*

Additions of information to specific fields of entity descriptions (e.g., attributes, steps, constraints, etc.) form the basis of other modification heuristics.

Cues from the discourse manager, such as the topic of discourse or recently referred to entities, are the key to K^{nAc} 's discourse heuristics. Because the current system lacks a sophisticated discourse manager, the only discourse heuristics being used are:

Heuristic D1: *Entities close to specified topics are likely to be referenced or modified.*

Heuristic D2: *Referenced entities are likely to be modified or referenced again.*

These heuristics generate expectations of some unspecified modification to the referenced entities or to those semantically close to them. "Closeness" is determined by the number and types of relationships (i.e., links) separating two entities.

4.2 Managing Expectations

As the number of expected modifications to the knowledge base grows, K^{nAc} 's ability to use the expectations to focus its attention diminishes. Thus, a means of selecting the most likely expectations (for a given time) is required. This is accomplished by assigning a rating to each expectation and pruning the set of heuristics based on this rating.

Each heuristic is responsible for determining a rating for each expectation it generates. This rating depends on the quality of the data and the specificity of the heuristic. In addition, each heuristic assigns a function to its expectations that specify how these ratings will change with time. For instance, certain expectations are important when they are created but become less valid with the passage of time; others become more critical as time passes. Some become more (or less) significant based on some state of the knowledge base; others are always valid. The functions currently available in K^{nAc} are: *fade*, *increase*, *until*, *while*, *after*, *for*, *always* and *never*.

5 Status and Conclusions

In this paper we have examined an often overlooked aspect of the knowledge acquisition process: the assimilation of information presented by a domain expert into an existing knowledge base. Though a fundamental part of the current conventional knowledge base development process, the issue of automatically locating and appropriately modifying existing knowledge to conform to the domain expert's descriptions has received little, if any, emphasis. Most current knowledge acquisition tools place this burden on the domain expert, forcing him to take over part of the knowledge engineer's task. By automating this assimilation process, the K^{nAc} system better insulates its user from the knowledge base.

K^{nAc} accomplishes this assimilation by: 1) comparing entity descriptions provided by the domain expert with existing knowledge base descriptions, 2) evaluating these matches in the context of the knowledge acquisition discourse, 3) making the modifications to the existing descriptions implied by the expert's information, and 4) generating (and managing) expectations of further changes to the knowledge base.

This work has developed several generic matching (and match evaluation) techniques especially adapted for knowledge acquisition. They shift the focus of matching from examining how closely two entities match to exploring the likelihood of their being modified so as to match. This is accomplished by matching procedures (for sets of entities and collections of constraints) which determine differences as well as similarities, an evaluation technique which explores the probability of the modifications required to make entities match and the degree to which these modifications are expected, and a means of anticipating modifications to the knowledge based on heuristic information about the knowledge acquisition process.

The K^{nAc} system is implemented in Common Lisp running on a TI Explorer®. Its current use is still experimental, though it has been able to assimilate a 20 step dialog on the travel reimbursement process, correctly constructing an internal representation of the plan in the POISE knowledge base. A complete description of the implementation and the

sample dialog can be found in [Lef87].

References

- [CL84] W. Bruce Croft and Lawrence S. Lefkowitz. Task support in an office system. *ACM Transactions on Office Information Systems*, July 1984.
- [CLLH82] W. Bruce Croft, Lawrence S. Lefkowitz, Victor R. Lesser, and Karen Huff. POISE: An intelligent assistant for profession based systems. In *Proceedings of the Conference on Artificial Intelligence*, Oakland University, Michigan, April 1982.
- [EEMT86] Larry Eshelman, Damien Ehret, John McDermott, and Ming Tan. MOLE: A tenacious knowledge acquisition tool. In *Proceedings of the Knowledge Acquisition for Knowledge-Based Systems Workshop*, Banff, Alberta, Canada, 1986.
- [GWP85] Allen Ginsberg, Sholom Weiss, and Peter Politakis. SEEK2: A generalized approach to automatic knowledge base refinement. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pages 367-374, 1985.
- [KC86] *Knowledge Craft Users Manual*. Carnegie Group Inc., 1986.
- [KNM85] Gary Kahn, Steve Nowlan, and John McDermott. MORE: An intelligent knowledge acquisition tool. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pages 581-584, 1985.
- [Lef87] Lawrence S. Lefkowitz. *Knowledge Acquisition through Anticipation of Modifications*. PhD thesis, University of Massachusetts, Amherst, MA, 1987.
- [Tve77] A. Tversky. Features of similarity. *Psychological Review*, 84(4):327-352, July 1977.
- [WF83] M. Wright and Mark S. Fox. *SRL 1.5 User Manual*. Intelligent Systems Laboratory, Carnegie-Mellon University Robotics Institute, 1983.

The Role of Plan Recognition in Design of an Intelligent User Interface*

Carol A. Broverman Karen E. Huff
Victor R. Lesser

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

July 17, 1986

Abstract

We report on the design and implementation of intelligent interface that assists users performing computer-based professional work by recognizing sequences of actions that are globally consistent and meet desired goals. Our approach is based on hierarchical plans that represent user tasks. Recognition of instantiations of these plans occurs by predicting future actions from past events and then matching new actions to these predictions. The intelligent interface GRAPPLE (Goal Recognition and Planning Environment) extends a previous system primarily through a reformulation of the plans, incorporating more knowledge about the plans and the domain. We present this new formalism, which lays the groundwork for the development of meta-plans and reasoning from first-principles.

1 Introduction

In complex domains of computer-based professional work, there is a need for intelligent interfaces which assist practitioners (as opposed to novices) with sequences of actions which meet desired goals and are consistent in their global context. It is not

*This work is being supported by the Air Force Systems Command, Rome Air Development Center, Griffiss Air Force Base, New York 13441-5700, and the Air Force Office of Scientific Research, Bolling Air Force Base, District of Columbia 20332, under contract No. F30602-85-C-0008.

a question of replacing the practitioner with an expert system, but rather of cooperatively supporting the work of the practitioner with an intelligent assistant. This assistant would bridge the gap between the practitioner's perspective on problem-solving activities and the computer system's perspective on tool invocations and resource usage.

Using predefined, hierarchical activity definitions (plans), the intelligent interface can monitor the conversation between user and computer system, recognizing the commands issued as instantiations of primitive plan definitions. Predictions of expected user commands can be made as a result of the successful integration of a primitive action into a more abstract plan representation. The interface can also operate in an alternate mode and automatically generate sequences of primitive commands in response to a user request for a high-level plan. Such an intelligent interface would thus be a mixed-initiative system, combining facilities for plan recognition and plan automation, with an embedded planner used to extend the predefined plans as needed. The assistance provided to users would include:

- Detecting actual and potential errors, including errors of global strategy;
- Recovering from and correcting errors using context and goal information;
- Creating and managing agendas of work to be performed;
- Summarizing accomplishments of terminal sessions;
- Automatically performing steps in a plan or completing a plan.

Realizing this type of intelligent interface will require AI techniques for several reasons. During recognition, the information needed for a definitive interpretation of a plan may not be complete. Further, the plan definition may be approximate and based, in part, on heuristic knowledge. The interface will have to generate selected alternatives, make choices, and be prepared to retract interpretations at a later time. It will usually be too burdensome to write predefined plans to cover all possible situations, so there will be a need to generate new plans dynamically (either to interpret user actions or to carry out some high-level plan). An embedded planner will ensure robustness of the interface over a wider range of activities. Finally, a deep knowledge of the domain, together with appropriate automated reasoning techniques, will add to the power of all aspects of the interface.

In this paper, we discuss a research effort to investigate the role of plan recognition in the design of intelligent interfaces. A first-generation system, called POISE (Procedure Oriented Interface for Supportive Environments) is briefly presented as an intelligent interface which incorporates plan recognition viewed from an

event-based perspective. The limitations of this approach are cited, and a second-generation system is being designed and implemented to extend the earlier effort. This new goal-based system, called GRAPPLE (Goal Recognition and Planning Environment), is presented in detail. Fundamental changes and the resulting capabilities are highlighted. The last section of this paper is devoted to a discussion of current research directions, such as the incorporation of meta-plans, and the representation and use of first principles knowledge about the domain.

2 An Event-Based Intelligent Interface

We have implemented an intelligent interface called POISE [1,2,5] which performs simple plan recognition and plan automation in the domain of office automation. POISE is written in Common Lisp and is integrated with DEC FMS office automation tools and VMS mail facilities.

The original POISE system successfully addresses the goal of implementing a mixed-initiative intelligent interface which recognizes and automates plans which are represented in a hierarchical plan library. The plans POISE uses are behavioral (event-based) in nature; an abstract plan is decomposed by specifying a combination of lower-level plans, using grammar rules with temporal operators. These temporal operators include the regular operators of concatenation, alternation, and repetition, supplemented with a few concurrency operators[4]. Thus, an ordering of subplans is specified explicitly within the more complex plan, and this ordering constitutes the skeletal definition of the abstract plan.

A semantic database is accessed by the POISE interface to model all domain objects being manipulated by the plans. Objects are represented using a frame-based language[1], exploiting the inheritance feature for ease of specification, and including facilities to represent semantic constraints within and between objects.

The POISE system provides solutions to the problems of incomplete information and search complexity through its focusing mechanisms. A set of heuristics is used to limit the generation of alternatives when ambiguity arises, and truth-maintenance techniques are applied to retract incorrect interpretations [2]. Constraints from both the semantic database and within the plans are propagated statically and dynamically throughout the active instantiation network. Constraint propagation results in further pruning of predicted user actions, a limited capacity for error detection, and an ability to automate the completion of a partially instantiated plan.

2.1 Limitations in the First Generation System

POISE is limited in the types of reasoning which can be performed about relationships among plans. The behavioral plan definitions specify decomposition solely in terms of temporally ordered subplans and do not represent plan preconditions or goals. This characterization of a plan is insufficient for a planner whose role is to synthesize new plans, since there is no representation of the reasons behind the substeps, and little knowledge is available about the relationships between different plans. For example, a POISE plan may require that subplan B follow subplan A, but it does not describe the semantic database state established by subplan A and required by subplan B. Also, the temporally-ordered substep form of plan decomposition is rigid and lacks the modularity needed to easily integrate new plans into the plan library. Since environments can be arbitrarily complex and dynamic, limitations imposed by a purely event-based representation are significant.

Also, without explicit goals, there is no way to note that an action may be omitted because its goal has already been met. Nor is there enough knowledge to support a robust approach to recognition and recovery from plan failure. With an event-based approach, recovery must be built into the plan rules, making the rules unwieldy and complex and discouraging deeper reasoning about failure and recovery. Since in most semi-structured environments, the "main-line" or standard definition of a plan is actually less common than the variations and exceptions which occur [7], a serious attempt to overcome these limitations imposed by the event-based approach should be made.

Another limitation of POISE is inherent in the way the various types of knowledge are represented. In POISE, the domain knowledge is expressed using a frame-like model of domain objects, while the plan knowledge is represented using a completely different formalism and underlying representation tool. The use of a uniform representation for both domain plans and domain objects would allow the system designer to tie together constraints related to both plans and objects, and provide greater coherency[1]. In addition, general domain knowledge that is not directly related to either a plan or object definition is not represented in POISE. Thus the knowledge needed for a deeper model of the domain is lacking, seriously limiting reasoning that can be done about plan failure or while handling exceptions that arise during plan recognition.

3 A Second Generation System

We are currently designing and implementing a successor to POISE called GRAPPLE. This system is being developed in order to address shortcomings inherent in the original POISE system, and to pursue further problems which arise when performing plan recognition in a largely unstructured domain. We are particularly interested in exploring potential sources of deeper domain knowledge than those exploited by POISE, thus motivating a reevaluation of the characterization and interpretation of plans. As a testbed for GRAPPLE, we are using the domain of software development, which is a complex domain and offers rich sources of knowledge, yet is relatively self-contained.

3.1 Fundamental Changes

An overriding theme of the GRAPPLE plan formalism is an expanded representation of knowledge about plans and their interrelationships. The *goal* of a plan and the *effects* of a plan on the domain model are explicitly represented, as are *preconditions*, which must be satisfied before the plan can be executed. A deeper representation of the plan increases the capacity for reasoning during plan recognition and automated planning and affords the system a much richer knowledge base from which to reason about plan failure. The system also has a larger store of semantic knowledge which it can use to "understand" and accommodate exceptional scenarios during plan recognition.

The use of a state-based, goal-oriented perspective in GRAPPLE is in contrast to the POISE event-based substep plan characterization and follows the classical planning formalism. A *goal* is specified as a partial state of the semantic database. A goal can be decomposed into *subgoals*, each of which also is expressed as a semantic database state specification. Achievement of all the subgoals, along with the posting of the *effects* of the plan, should lead to satisfaction of the goal of the plan. *Effects* can be expressed in high-level as well as primitive plans, allowing for the expression of complex semantic changes to the semantic database.

A state-based approach to plan representation provides the system more modularity. For example, if one of the subgoals for a plan is to *have-more-disk-space*, a number of plans may be retrieved that achieve this subgoal; for instance: *delete-a-file*, *purge-directory*, and *increase-quota*. The multiple possible plans need not be specified statically; they can be determined dynamically in order to exploit the rich sources of contextual knowledge at runtime. Representing goals as states in GRAPPLE also allows the interface to avoid a potentially redundant execution of a plan. If a plan has a subgoal which is already satisfied, then no plan need be

executed to achieve the subgoal. The overall ordering of the plans that can achieve subgoals of a complex plan is determined dynamically by monitoring the satisfaction of *preconditions*. The state-based approach thus allows for the easy addition and removal of plan definitions from the plan library, without necessitating a recompiling of all the plans and their subgoals. In POISE, the event-based plan specification is "hard-coded," thus rendering the plan library inflexible to dynamic modifications.

GRAPPLE also attempts to overcome limitations imposed by POISE's nonuniform representations. In GRAPPLE, plans are represented with the same knowledge representation tool/language as domain objects. Therefore relationships between certain plans and objects can be easily recorded and constraints relating to both plans and associated domain objects are uniformly specified. The groundwork is thus laid for a more powerful object representation language and more powerful reasoning capabilities.

In order to provide complete coverage of relevant activities, GRAPPLE also models objects and processes which are not directly monitorable. Certain actions, such as decision making by the user, always occur "offline." Other actions, such as communication, may occur "online" through mail facilities or "offline" via phone or in person. Even when "offline," these actions cannot be ignored in constructing a total picture of the user's activities; for example, such actions may satisfy the preconditions of later actions. To handle this, any plan can be denoted "offline", in which case GRAPPLE must deduce when its execution has occurred and when its effects should be posted to the semantic database.

3.2 GRAPPLE Plan Formalism

A plan definition in GRAPPLE consists of a set of clauses: *Plan-name*, *Goal*, *Plan-Vars*, *Builtin-Vars*, *Precondition*, *Subgoals*, *Constraints*, and *Effects*. Plans may be either "builtin" or "complex". "Builtin" plans are those plans which map directly to primitive commands that may be issued by the user in the programming environment and will have a *Builtin-vars* clause but will never have a *Subgoals* clause. "Complex" plans, by definition, require multiple subplans to achieve their goals. Each intermediate step corresponds to a subgoal in the *Subgoals* clause. Plan may also be "offline" if they model user decision-making or some other non-monitorable activity. All types of plans have a *Goal* clause, but the other clauses do not have to be present in the plan definition, except where just mentioned.

The *Goal* clause identifies a state¹ of the semantic database that is achieved by

¹Obviously, this as well as other semantic database state specifications are *partial* states of the semantic database, since it deals with only a few aspects of the entire state, which is the conjunction of every fact in the semantic database.

the successful completion of the associated plan. The goal is expressed as a predicate calculus proposition whose truth can be determined by querying the semantic database. The *Goal* of a plan is distinct from its more abstract *purpose*, which is determined dynamically by the integration of an instantiation of this plan into a hierarchical interpretation at runtime.

The *Builtin-vars* clause is present only for "builtin" plans, and defines the primitive values that are determined by the filter program². The *Plan-vars* clause defines the names and types of the input and output parameters of the plan. The directional flow of the parameters is determined by the goal statement; those parameters which are bound in the *Goal* are the output parameters.

The *Precondition* clause defines the initial state of the semantic database that must hold in order for the plan to be allowed to begin. It is expressed as a proposition in predicate calculus. The *precondition* may be "locked," in which case, once it is achieved, it cannot be negated until the plan actually begins³.

The *Subgoal* clause is present for "complex" plans, and consists of a set of semantic database states, again expressed as predicate calculus propositions. The *Subgoals* represent the decomposition of the plan *Goal*. Thus, complex plans are not defined in terms of other plans, but indirectly through states of the semantic database which may be achieved by other plans. Individual subgoals in the *Subgoals* clause may also be "locked," which indicates that once achieved, a subgoal must persist until the completion of the entire plan. In general, though, it is not required that all subgoals be true at the completion of a plan. The order in which subgoals are to be achieved is determined dynamically, dictated by the *preconditions* of the plans chosen to accomplish them. A notation is provided for denoting "iterated subgoals", indicating that a subgoal may be achieved repeatedly with different variable bindings while completing the plan.

The *Constraints* clause specifies constraints that must hold within and between variables used by any of the clauses of the plan. They are expressed as predicates on the semantic database.

The *Effects* clause specifies modifications that are to be made to the semantic database upon completion of the plan. New objects can be created, and attributes and entities can be added or deleted. Additions and deletions from the semantic database are specified as predicates and qualified by the type of modification (ADD or DELETE). New entities are specified with the NEW qualifier. All types of plans

²The filter program monitors user actions, and traps all command issued by the user. It is responsible for determining the type of "builtin" plan that corresponds to the command, and presenting the primitive parameters of that invocation to the intelligent interface in a standardized form.

³The start of a plan is defined by either the occurrence of a primitive action which is integrated as a subpart of that plan or the occurrence of the plan itself, if "builtin".

may have an *Effects* clause, allowing the expression of a complex, high-level change to the semantic database.

The semantic database, used in POISE to model the world of the user, serves the additional role in GRAPPLE as the state description of a classical planning system. It is consulted to determine if a goal, precondition, subgoal, or constraint is true. The effects clause serves as a non-procedural description of a state transition. The semantic database may be thought of as an entity-relationship model, with entities, attributes of entities, and relationships between entities. The usual translation to predicate calculus notation may be made, whereby the entities become constants, the attributes map into functions, and the relationships map into predicates and (optionally) additional functions[3]. These constructs are then used in the sentences of which the various plan clauses are composed. An example GRAPPLE plan definition is given in Figure 1, and a portion of the semantic database, with links and attributes corresponding to available predicates and functions, is shown in Figure 2.

3.3 Plan Recognition in GRAPPLE

The plan recognition component of GRAPPLE is currently being designed and implemented. Basic mechanisms have been established for predicting expected actions based on occurrences already seen and for incorporating an occurrence of an action into an interpretation structure. An *expected-actions* list is maintained for each top-level plan to record the monitorable user actions predicted by the interface. A *pending-conditions* list is also associated with each top-level plan to record those *goals, subgoals, and preconditions* that are awaiting satisfaction.

At any point in time during the running of the intelligent interface, there are one or more top-level plans which are in progress. They are represented by instantiations of those plans on the *active plan blackboard*. When a plan is instantiated, each of its goals and subgoals is instantiated as well and maintained as *pending conditions* for that plan. A backward-chaining approach is then taken to predict which plans could achieve these *pending conditions*.

Predictions are currently⁴ made by matching the subgoal/goal conditions with the goals of other plans in the plan library. Once a prediction is made, an instantiation structure is created for the predicted plan and its precondition is posted to await satisfaction. If the plan is a primitive one, it is posted to the list of *expected-*

⁴A more complete and sophisticated prediction mechanism will be incorporated upon the addition of a more sophisticated *planner* module, which will analyze the interactions between *effects* of plans and pending *goal* conditions.

(PLAN-NAME *edit* IS-COMPLEX

PLAN-VARS: (t : text; ds : abstract-spec)
GOAL EXISTS (t) |
latest-realization(t, ds) AND believed-consistent(t,ds)
PRECOND EXISTS (b: text) | baseline(b,ds)
SUBGOALS (NAME Accessible)
(VARS (f : file) (t : text))
(STMT (EXISTS (f) | stored-in(t,f)))
(NAME Created)
(VARS (nt, ot : text) (ds : abstract-spec))
(ITERATED)
(STMT (EXISTS (nt) |
latest-realization(nt,ds) AND successor(nt,ot)))
(NAME Accepted)
(VARS (t:text) (ds : abstract-spec))
(COMPLETES Created)
(STMT (believed-to-be(t,ds))
CONSTRAINTS (Accessible.t = Created[first].ot)
(Created[this].nt = Created[next].ot)
(Created[any].ds = ds)
(Accessible.t = b)
(ds = Accepted.ds)
(Created[final].nt = Accepted.t)
EFFECTS *None - accomplished by subplans*

Figure 1: A GRAPPLE Plan Definition

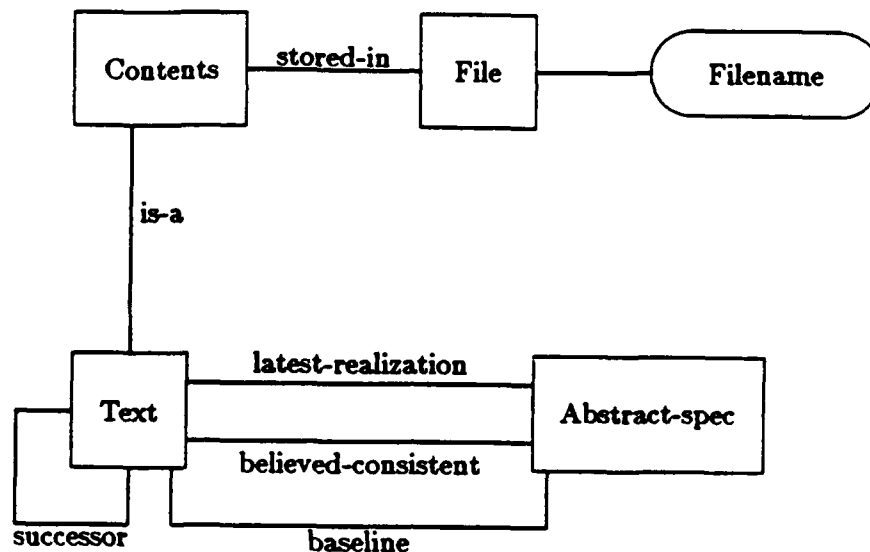


Figure 2: Subset of Semantic Database in GRAPPLE

actions for the top-level plan that subsumes it.

When a user-action occurs, a *matcher* is invoked to determine which of the *expected-actions* is being performed. Values determined by the filter program, which directly monitors user actions, are passed up to the designated expected action structure, and bindings of variables are propagated. *Pending-conditions* are reevaluated and the plan recognizer generates new expectations after integrating the action occurrence.

Choice points have been identified at various stages during plan recognition. For example, one choice point occurs when many plans qualify as achievers of an outstanding goal or subgoal. Another arises when an incoming action matches several predicted actions on the *expected-actions* list and there is not enough information to disambiguate. Heuristics to guide the focus of control and to limit search are currently under investigation.

3.4 Current Research Directions

Work is in progress to develop a model for describing and using meta-plans, which are special plans describing the use of the domain plans, and to explore the potential for reasoning using first-principles knowledge about the domain.

3.4.1 Meta-Plans

We are currently working on a meta-plan approach to provide more types of relationships between plans, in addition to subgoal decomposition. Recognizing plan failure and integrating the resulting recovery actions are particularly important in domains like software development, where the basic paradigm of work is "trial and error." Also, during informal analysis of programmer terminal sessions, we have noticed other plan interrelationships. Obtaining on-line help provides the user with specific information to be able to formulate and issue some other command. Gathering information via tools to analyze, reorganize, condense, and present data supports the user in making key decisions about how to carry out some plan. At times, programmers will model a plan with dummy input in order to see if it will work as they predict. Occasionally, work is undertaken in experimental mode, where the initial state is explicitly saved in advance, the work then performed, and a decision made as to whether to accept the results or back-up to the initial state and try again.

Meta-plans allow us to capture these general patterns as a context for executing any plan, without having to write out all the details in every plan. In our work with meta-plans to date, we have found that the same basic plan formalism with goal, precondition, subgoal, constraints, and effects clauses can be used. The meta-plan variables are not domain objects, rather, they are domain plans, their goals, effects, etc. While it was not one of our original goals, we found that meta-plans can be written so that the effects manipulate the actual recognition data structures described in section 3.3. Thus, we can implement the intelligent interface at the topmost level as a simple plan execution system, where execution of the meta-plans causes recognition of the domain plans.

3.4.2 Incorporating First Principles Knowledge

As we have worked with plan definitions of either the state-driven or event-driven type, we have recognized that there is additional knowledge about the domain which is not appropriately expressed in the plans themselves. This is particularly true in specialized domains such as software development, where there is a rich set of technical concepts (such as versions, history, configurations, properties and bugs of modules) and a broad range of *first principles* knowledge about programming. This knowledge forms a self-contained world for reasoning about actions, and will, we believe, be an important addition to the intelligent interface.

This first principles knowledge can be used in the intelligent interface in several different ways, to improve interface performance and extend more assistance to the

user. Using the first principles knowledge to generate tentative bindings of plan parameters will result in earlier, more detailed prediction, and will also limit the number of alternatives to consider during recognition or execution of plans. It provides an alternative to simple heuristics such as "prefer the continuation of a plan already in progress to the start of a new plan" for choosing among alternatives, which may be increasingly important as the number of alternatives grows or when plans are inherently underspecified. It can be used to double-check decisions made by the programmer (modeled in the offline plans). Finally, first principles knowledge can provide additional semantic distinctions between apparently equivalent actions (fixing a bug versus adding a new feature) so that future programmer decisions (such as what tests to run) can be anticipated and double-checked.

4 Status

We have defined the GRAPPLE plan and semantic database formalism and are currently completing the plan recognition algorithms, including constraint handling and focusing. Knowledge Craft [6], a knowledge representation tool package that offers a logic programming environment built on top of a frame-based knowledge representation, is being used to implement the system. A large set of plans for a Unix⁵/C software development environment has been written in the GRAPPLE formalism, and we are starting to formalize the first principles knowledge for this domain. We have also started work on appropriate meta-plans in order to provide integrated interpretations for the entire spectrum of user actions.

References

- [1] Broverman, C.A.; Croft, W.B. "A knowledge-based approach to data management for intelligent user interfaces," Proceedings of Conference for Very Large Data Bases 11, Stockholm Sweden, 1985, pp.96-104.
- [2] Carver, N.; Lesser, V., McCue, D. "Focusing in Plan Recognition," Proceedings of AAAI, Austin, Texas, pp.42-48, 1984.
- [3] Chen, P.P. "The entity-relationship model: toward a unified view of data," *ACM Transactions on Database Systems*, 1:1, pp.9-36, March 1976.

⁵Unix is a trademark of AT & T Bell Laboratories.

- [4] Croft, W.B.; Lefkowitz, L.S. "An office procedure formalism used for an intelligent interface." COINS Technical report 82-4, University of Massachusetts, 1982.
- [5] Croft, W.B.; Lefkowitz, L.S., "Task Support in an Office System," *ACM Transactions on Office Information Systems*, vol. 2, pp.197-212, 1984.
- [6] Knowledge Craft Manual Guide, Vax/VMS Version 3.0, Carnegie Group Inc., March 1986.
- [7] Kunin, J.S. Analysis and Specification of Office Procedures. Ph.D. thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Ma., February 1982.

APPENDIX 5-F

Planning Discourse in an Intelligent Tutor

**Beverly Woolf
David D. McDonald**

20 February 1986

**Department of Computer and Information Science
University of Massachusetts
Amherst, Massachusetts 01003**

Abstract

Tutoring is a highly constrained and specialized form of communication between a tutor and student. As such it requires complex discourse skills, knowledge of the domain, and knowledge of common errors. Building a machine tutor that replicates these discourse skills requires attention to the representation and manipulation of domain and discourse knowledge. This article describes a tutoring system that reasons about its choice of tutoring strategies, domain knowledge, and discourse conventions before engaging the student. The system can discuss two domains and has focused on the implementation of three computational elements: a discourse management network, pedagogical annotations of the domain knowledge, and qualitative inferences about the student and topic. This article discusses these three components, discourse systems in general, and theoretical issues in the implementation of man/machine interfaces, especially they impact on the design of systems for tutoring discourse.

1. Tutoring as Communication

Tutoring presents both tutor and student with fundamental problems of communication. For example, an effective tutor must recognize whether his message has been received by the student, whether it was understood, and how the very form of that message could have influenced the student's answer. The student, for his part, must recognize if data delivered by the tutor has been consistent with his own "understanding" of the domain and of the conversational situation. He must judge whether the tutor's teaching strategies or choice of topic provide insight into saliency or relational information that he might be unaware of.

Compounding this situation is the fact that the student generally does not know what it is that *he does not know* and one of the goals of the tutor's discourse is to clarify this data. Tutoring discourse, therefore, employs specialized rules and complex strategies to facilitate an adroit tutor in knowing how to talk, how to listen, what to know, and how to teach.

Building a machine tutor capable of replicating these specialized rules and complex strategies requires careful representation of both domain and discourse knowledge. We are engaged in building a system, named *Meno-tutor* [1, 2], that addresses these issues; our research is also being applied in several natural language and interface areas, including legal reasoning and scheduling. The system employs a discourse controller and associated knowledge bases to make decisions about the choice of tutoring strategies, domain topic, and discourse move. Each decision is based on the context of the student's presumed knowledge and the current history of the discourse history. We call this kind of system "context-dependent" because its output behavior is generated only after consideration of discourse interactions and in relation to the nature of student knowledge and expert knowledge.

Meno-tutor teaches in two different domains, rainfall and Pascal programming, and predicates its choice of topics on consideration about tutoring strategies and communication skills. The article discusses three components that are used by Meno-tutor to structure the generation of discourse. The first is a Discourse Management Network that provides a general framework within which tutoring rules can be defined and tested. The second is a system of pedagogically motivated annotations on the knowledge base that measures a student's presumed knowledge per topic and orders topics according to the system's perception of the learner's ability. The third component is a model of discourse conventions and qualitative reasoning about discourse that used to refine discourse decisions.

1.1 The Nature of Tutoring

We define tutoring as communication between a tutor and a student whose goal, in general, is to clarify a body of knowledge to which the student has already been exposed (e.g., through lectures or reading). Though tutoring can be non-verbal or graphic, it typically involves an extended linguistic exchange in which the tutor engages the student in effective discourse even in the face of student errors. Not every activity we might casually associate with "tutoring" is, in fact, effective as a tutoring tool, for example, pre-tests, post-tests, and drill and practice exercises. These have developed in the context of classroom teaching where the student/teacher ratio is large and one-on-one, student-specific tutoring has not been possible. These activities do not represent essential vehicles of tutoring. One of our goals in developing competent machine tutors is to reduce this large ratio of students to tutors and to establish the more sensitive one-on-one interaction as the norm.

In addition to worrying about the nature of tutoring, we also must address discourse issues in general. *Human* speakers employ subtle linguistic cues to shift topics or provide supplementary knowledge; *human* listeners use these cues to set up expectations about the underlying structure of the discourse. The expectations set up by listeners are what the speaker tries to anticipate and to deliberately control for. Such expectations and inferences are powerful forces in structuring any conversation; they are especially powerful in tutoring.

For example, a tutor might choose a conversational move based on the student's actual responses to his question or he might choose a response based on his interpretation of that response and his "reading" of the student's knowledge based on that response. For instance, the tutor might make assumptions about the student's lack of knowledge, as for example when a sequence of wrong responses implies that the student doesn't know the answers. (He may not have understood the question.)¹ We are researching discourse expectations in order to anticipate a student's *choice* of responses and to formalize discourse expectations into computational elements, such as plans and rules. The machine we have built and the extensions we are currently building respond to broad inferences about a student's knowledge, to his model of the domain, and to the current discourse history. The basic problem is how to make inferences about the user and how to have these inferences govern the form of the text produced.

¹ "We recognize that a machine cannot know with certainty what a listener knows, neither can a listener know what a machine knows; a machine cannot be omniscient or clairvoyant. However, the machine can deduce, on the basis of evidence, something about what the listener assumes. These assumptions can be used to govern the form of the text generated. A primary goal of machine discourse is to clarify deductions made about the listener by bringing about situations (e.g., by the questions it asks and the choice of information it supplies when correcting or explaining) that facilitate these deductions.

Tutor: Do you know what the climate is like in Washington and Oregon?

Student: Is it cold?

Tutor: No, it is rather mild. Can you guess about the rainfall there?

Student: Normal, I guess.

Tutor: Well, the Japan current, which starts in south eastern Pacific, goes along the coast of Japan and across the North Pacific ending up off the coast of Washington and Oregon. How do you think that current affects the climate there?

Student: It's probably rainy.

Tutor: It is indeed very wet, there are rain forests in both Washington and Oregon. What does that tell you about the temperature of the Japan current?

Student: It's warm.

Figure 1: A Human Discourse from Stevens [3].

student answered the next two questions about "rainfall" and the "temperature of the Japan Current" correctly.

1.3 Tip of the Iceberg in Discourse

There are interesting problems to be solved before a machine tutor can be designed that displays the amount of flexibility shown by this example. One of them is how to recognize a person's unspoken problems or intentions. We call this the "tip of the iceberg" problem, by which we mean that the words spoken by the student or the teacher are only a small portion of what they know about each other and what they are communicating in a discourse. Some of the first work on this problem was done by Allen [4], who built a system to model the knowledge needed by an effective agent at a train station. The

For instance, the adjustments that a computer tutor makes should be dependent upon its specific experience with a specific student. A variety of experiences should lead to a variety of responses such that machine response to a knowledgeable student is fundamentally different, both in style and content, from the same system's response to the confused student.

Further, a computer tutor should not simply correct false answers; in the case of a student's wrong answers for example, it should resolve issues such as:

>> whether it is preferable to explain the error or to start a lengthy exploration of the student's knowledge;

>> whether to allow uncertainty about the student's knowledge to persist temporarily while it explores a potential misconception;

>> how hard it should work to understand why a student answered a question incorrectly or how much effort should be exerted to resolve questions about the student's presumed knowledge or misconceptions; and

>> when and how to explain a wrong answer given if that is the best response to make.

1.2 Human Tutoring

As an example of what an effective *human* tutor can do, we present a protocol of an expert human tutor working with a student on understanding rainfall in Figure 1. The protocol is taken from an earlier investigation of human tutoring behavior [3] and shows how the expert remains responsive and sensitive to the student's minimum knowledge. We suggest that the tutor began to question the student about general topics (e.g., "climate" and "rainfall") in an attempt to assess the student's frontier of knowledge. Then, reacting to the student's failure to answer the first two questions correctly, we believe that the tutor decided to change strategy to one that would provide the student with additional information which might help him infer the correct information. The strategy worked—the

discourse in Figure 2 is in the style of Allen's examples and is predicated on the agent anticipating or predicting the traveler's problem.

Traveler: "Is this the gate for the train to New York?"
Agent: "No, you want gate number 44 and the train leaves at 6:33."

Figure 2: Discourse to Handle Unspoken Problems.

If the traveler had been correct, then just saying "yes" would have been enough for the agent to do and the response would have solved the traveler's problem. However, the traveler was wrong and a simple "no" by itself *would* have been insufficient because it would not have addressed the traveler's actual problem. By giving the additional information, the agent showed that he recognized that what the traveler was *saying* was only a small part—only the tip of the iceberg—of what he actually intended to communicate.

The "tip of the iceberg problem" in tutoring is to recognize the student's unspoken confusions and misconceptions. By maintaining a record of the student's previous errors or by directly questioning him about his misconceptions, a machine tutor can acquire the same kind of information that our train agent used.

2. Overview of Meno-tutor

A major thrust of our research has been to develop the control and data structures needed to plan responsive discourse, such as that observed in human tutoring. In this section we discuss Meno-tutor [1] and identify the control structure used to customize the tutor's discourse for the individual student. Meno-tutor is a "generic" tutor, i.e., it is not committed by design to a single tutoring approach or tutoring domain. Rather, it provides

a general framework within which tutoring rules can be defined and tested.

We contrast our work with older tutoring and discourse systems [5, 6, 7, 8] that were "retrieval-oriented." While we have placed our emphasis on choosing among alternative responses that guide the learner based on what the tutor knows about him, other systems have placed emphasis on retrieving a correct answer. They sought to produce a correct answer independent of the user's knowledge or current history [9, 10]. More recent interface and tutoring systems [9, 10, 11] have begun to tailor their responses to the user and to discourse context.

Tutor: Do you think it rains much in Oregon?
Student: No.
Tutor: Why do you think it doesn't rain much in Oregon?
Student: Because the neighboring states are dry.
Tutor: The neighboring states are dry, but, you can not generalize from that. In fact, it does rain a lot in Oregon. Can you guess what causes the rain there?
Student: Well, there are high mountains in the region and the states border the Pacific Ocean.
Tutor: Yes. The Pacific borders Oregon. How do you think it is involved in the heavy rainfall there?
Student: Ocean air rises and cools upon reaching the land
Tutor: That is right.

Figure 3: Discourse Using an Alternative Tutoring Style.

As an example of a discourse produced by the Meno-tutor, we present Figure 1. Meno-tutor produced the actual text and high-level transitions in this discourse (we describe how in Section 3). This first tutoring discourse and twelve other human discourses were used to "reverse engineer" the Meno-tutor. That is, we analyzed the common transitions and speech patterns used in these discourses and then defined the structures and knowledge necessary for a machine tutor to have a similar model of the student and to make the same transitions. For instance, the tutoring system in the discourse of Figure 1 recognized that the student made two wrong answers and it inferred that his knowledge was limited.¹ It then judged that the question-answer approach, which had been used until then, was ineffective and should be changed and that a new topic, the "Japan Current," should be discussed because it is a dominant influence behind the region's climate. The system decided to supply the additional data in a descriptive, rather than an interrogative style, because the student seemed confused and might profit from the addition of supplemental data. At this point, Meno-tutor is not a fully capable tutor for any one subject but rather a vehicle for experimenting with tutoring in several domains. Its knowledge of the two domains on which it has been defined is shallow.²

¹ "It's not that those answers were simply "wrong," rather that they reflect reasonable default assumptions about the weather in "northern states." An attempt to probe the student's default assumptions is made in the next discourse, Figure 4.

² * Meno-tutor has been developed without a full-scale natural language understander or generator. The conceptual equivalent of a student's input is fed by hand to the tutor (i.e., what would have been the output of a natural language comprehension system) and the output is produced by standard incremental replacement techniques. We have not yet worked with MUMBLE, our surface language generator, because we haven't yet invested in building a large enough knowledge base to make the linkup useful. Our intent is to develop a complex knowledge base, probably in the domain of Pascal, to extend the surface language generator to deal with the domain, and to build a simple natural language parser to interface with the student.

The first discourse was generated by Meno-tutor in such a way that all the decisions, topics, and tutoring styles available to the system were explicit and modifiable by the authors. Modifying discourse decisions allowed us to generate additional discourses and to move beyond the "reverse-engineering" in this first discourse. We have begun to explore the "tutoring space" our apparatus defined by varying the domain and the particulars of the rules. The discourse in Figure 4, for instance, is based on the same domain as the first, but is done in an alternative tutoring style, brought about by modifying the "meta-rules" that govern whether the tutor explores the student's frontier (Figure 1) or probes the student's misconceptions about the current topic as soon as the first mistake is made (Figure 4).

```
1 PROGRAM LESSON1(INPUT, OUTPUT);
2   VAR
3     SUM, GRADES, STUDENTS: INTEGER;
4     MEDIAN: REAL;
5   BEGIN
6     SUM:=0;
7     STUDENTS:=0;
8     READ(GRADES);
9     WHILE GRADES>0 DO
10      BEGIN
11        SUM:=SUM + GRADES;
12        STUDENTS:=STUDENTS+1;
13        GRADES:=GRADES+1;
14          should be READ(GRADES);
15      END;
16      MEDIAN:=SUM/STUDENTS;
17      WRITELN
18        ('THE MEDIAN GRADE IS', MEDIAN:3);
19    END.
```

Figure 4: A Student Pascal Program.

Two meta-rules were modified to achieve this second discourse. The first moves the tutor to change its tutoring strategy. In the earlier discourse, this rule was used conservatively; the transition would have been made only after several topics were completely discussed and the tutor had some confidence about the student's knowledge or lack of it. In this discourse, however, the rule was applied after a single incorrect answer, thus shifting the focus of the discourse abruptly at the beginning of the discourse.

Discourse Produced Using a PASCAL Data Base

Tutor: Do you know that GRADES, in line 9, is a control variable for the WHILE loop?

Student: Yes

Tutor: Does that mean that GRADES should be
a) Initiated before the loop?
b) Added into SUM within the loop?
c) Modified within the loop?
d) Both a) and c)

Student: C

Tutor: You seem to understand about the control variable of the WHILE loop. Now let's talk about some other command statements. What do you think READ (GRADES) in line 8 does with the value of GRADES?
a) Reads all values of GRADES at one time.
b) Reads in a single value of GRADES.
c) Assigns several new input values to GRADES.

Student: A

Tutor: Then do you think that GRADES := GRADES + 1 updates the next of those values?

Student: Yes

Tutor: There is your misconception. Adding one to a variable is a common way to step through a range of values. But it cannot apply here because your READ instruction in line 8 retrieves only a single value. You should replace line 13 with a second call to READ (GRADES).

Figure 5: Menu-tutor Discussing Errors from the Program of Figure 4.

The second modified rule causes the tutor to question the student about misconceptions. Typically, this rule is triggered after all topics are complete, either by the questions about them having been answered correctly or by the student having been corrected by the tutor. In the second discourse, however, the rule was modified to eliminate that requirement, with the effect that the rule was enabled after the student's first incorrect answer that is linked in Meno-tutor's model to a common misconception.

In addition to producing a variety of tutoring styles by changing the meta-rules, we explored the tutoring space in a second way. We used the same teaching mechanism as used for the rainfall discourse and substituted a new knowledge base, this time about elementary PASCAL looping constructs.³ We wanted to demonstrate the power of isolating reasoning about tutoring strategies from reasoning about the knowledge to be tutored. One reason for this was to see if the tutoring component could be interfaced with a different expert knowledge base and a different language generator and could then teach a new subject and even "speak" in a new language. If our modularization was effective we could combine a Pascal knowledge base and, say a Chinese language generator, with the tutoring component and the resulting system could interrogate a student in Chinese and teach him about programming in Pascal. The difference in domain and language realization should force no changes in the tutoring component, though of course it might be quite inappropriate in China to use the same mix and structure of tutoring strategies as in the English language version of the system.

³ Meno-tutor was originally developed as part of a larger research effort directed at building an on-line run-time support system for novice Pascal users [13, 14]. As a part of this effort, a Bug Finder was developed that detected run-time semantic errors in novice Pascal programs [15, 16] and passed this information on to Meno-tutor. The Bug Finder could identifier the type of error and the line numbers of related variables. It was used for four semesters on classes of several hundred students at the University of Massachusetts.

The program in Figure 4 was actually submitted by a novice programmer and the discourse in Figure 5 actually generated by the original Meno-tutor with changes to expert knowledge base as will be discussed in Section 4.

The changes required to produce each discourse are described in Woolf [1]. Though the number of discourses produced is still small, the fact that our architecture allowed us to produce varied but still quite reasonable discourse as we changed the particulars of just a few rules, substantiates the overall effectiveness of our design.

3. The Discourse Management Network

The first mechanism used by Meno-tutor to customize discourse to the individual student is the Discourse Management Network (DMN). Meno-tutor separates the production of tutorial discourse into two distinct components: the tutoring component which contains the DMN, and the surface language generator. The tutoring component makes decisions about what discourse transitions to make and what information to convey or query; the surface language generator takes conceptual specifications from the tutoring component and produces the natural language output. These two components interface at the third level of the tutoring component as described below. The knowledge base for the tutor is a KL-ONE network annotated with pedagogical information about the relative importance of each topic in the domain discussed in Section 4.

The tutoring component is best described as a set of decision-units organized into three planning levels that successively refine the actions of the tutor, Figure 6. We refer to the network that structures these decisions, defining the default and meta-level transitions between them, as a Discourse Management Network or DMN. The refinement at each level maintains the constraints dictated by the previous level and further elaborates the possibilities for the system's actions. At the highest level, the discourse is constrained to a

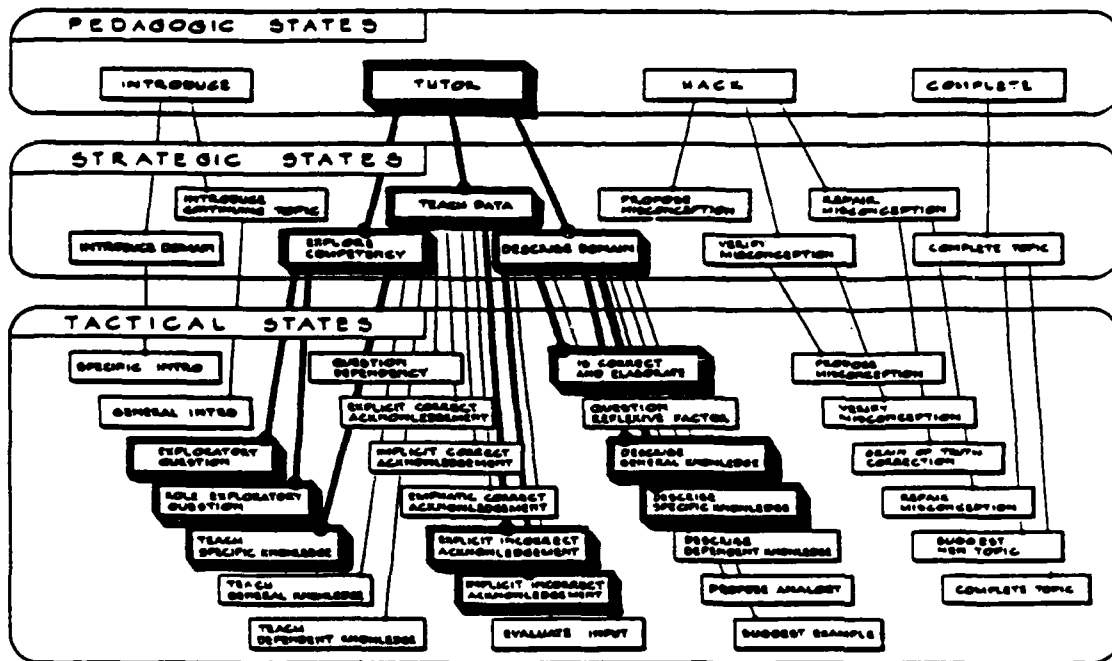


Figure 6: DMN used by the Tutoring Component.

specific tutoring approach that determines, for instance, how often the system will interrupt the student or how often it will probe him about misconceptions. At this level a choice is made between approaches which would diagnose the student's knowledge (*tutor*), or introduce a new topic (*introduce*). At the second level, the pedagogy is refined into a strategy, specifying the approach to be used. The choice here might be between exploring the student's competence by questioning him, or by describing the facts of the topic without any interaction. At the lowest level, a tactic is selected to implement the strategy. For instance, if the strategy involves questioning the student, the system can choose from half-a-dozen alternatives, e.g., it can question the student about a specific topic, the dependency between topics, or the role of a subtopic. Again, after the student has given

his answers, the system can choose from among eight ways to respond, e.g., it can correct the student, elaborate on his answer, or, alternatively, barely acknowledge his answer.

The tutoring component presently contains forty states, each organized as a LISP structure with slots for functions that are run when the state is evaluated. The slots define such things as the specifications of the text to be uttered, the next state to go to, or how to update the student and discourse model. The DMN is structured like an augmented transition network (ATN); it is traversed by an iterative routine that stays within a predetermined space of paths from node to node.

The key point about this control structure is that its paths are not fixed; each default path can be preempted at any time by a "meta-rule" that moves Meno-tutor onto a new path, which is ostensibly more in keeping with student history or discourse history. The action of the meta-rule corresponds functionally to the high-level transitions observed in human tutoring. Figure 7 represents the action of two meta-rules, one at the strategic and one at the tactical level. The ubiquity of the meta-rules—the fact that virtually any transition between tutoring states (nodes) may potentially be preempted—represents an important deviation from the standard control mechanism of an ATN. Formally, the behavior of Meno-tutor could be represented within the definition of an ATN; however, the need to include arcs for every meta-rule as part of the arc set of every state would miss the point of our design.

The system presently contains 20 meta-rules; most originate from more than one state and move the tutor to a single, new state. The preconditions of the meta-rules determine when it is time to move off the default path: they examine such data structures as the student model (e.g., Does the student know a given topic?), the discourse model (e.g., Have enough questions been asked on a given topic to assess whether the student knows it?), and the domain model (e.g., Do related topics exist?). Two meta-rules are described in an

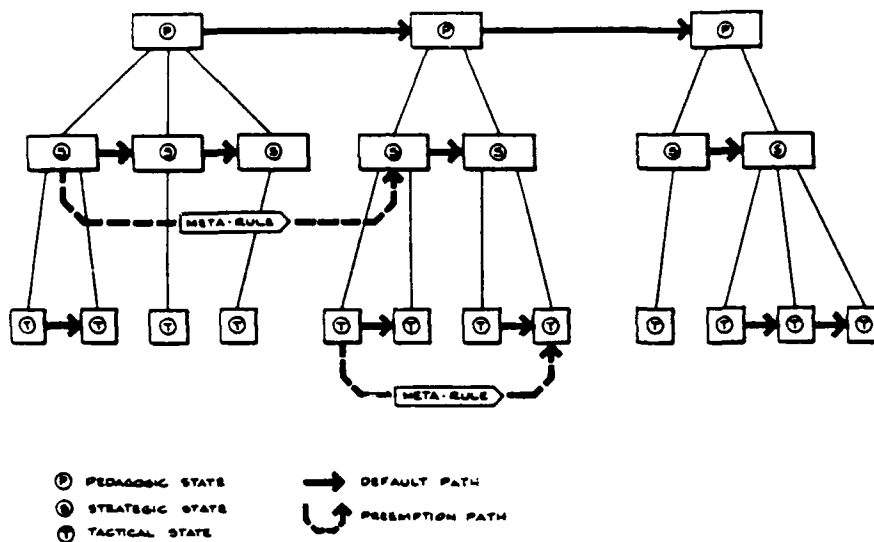


Figure 7: The Action of the Meta-Rules.

informal notation in Figure 8 and in more detail in the next section.

3.1 An Example of Discourse Planning

In this section, we provide an explicit view of the way the decision-units and meta-rules interact in the tutoring process. We describe the generation of a portion of the discourse given below.

TUTOR: No, it is rather mild. Can you guess about the rainfall there?

STUDENT: Normal, I guess.

TUTOR: Well, the Japan Current, which starts in the southeastern Pacific, flows along the coast of Japan and across the North Pacific, ending up off the coast of Washington and Oregon. How do you think that current affects the climate there?

The example begins after the student has already answered one question incorrectly. Figures 9 and 10 show Snapshots of Meno-tutor's passage through a small portion of the Discourse Management Network (DMN) as it plans and generates the sample discourse.

S1-EXPLORE - a Strategic Meta-rule

From: teach-data
To: explore-competency

Description: Moves the tutor to begin a series of shallow questions about a variety of topics.

Activation: The present topic is complete the tutor has little confidence in its assessment of the student's knowledge.

Behavior: Generates an expository shift from detailed examination of a single topic to a shallow examination of a variety of topics on the threshold of the student's knowledge.

T6-A.IMPLICITLY - a Tactical Meta-rule

From: explicit-incorrect-acknowledgement
To: implicit-incorrect-acknowledgement

Description: Moves the tutor to utter a brief acknowledgement of an incorrect answer.

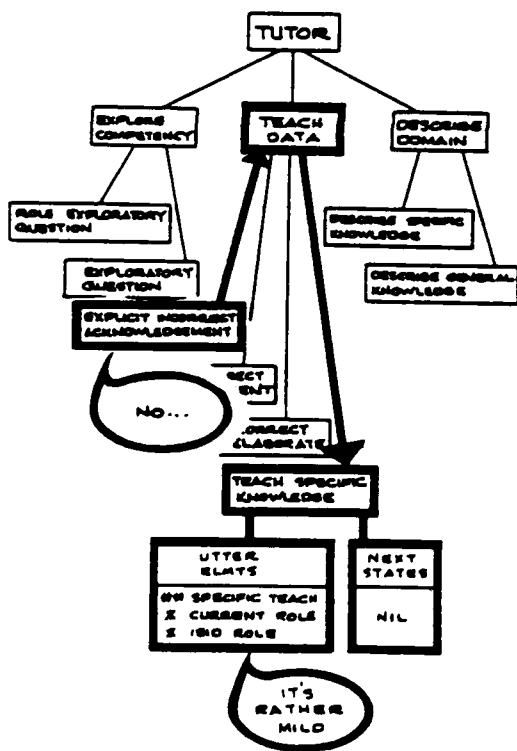
Activation: The wrong answer threshold has been reached and the student seems confused.

Behavior: Shifts the discourse from an explicit correction of the student's answer to a response that recognizes, but does not dwell on, the incorrect answer.

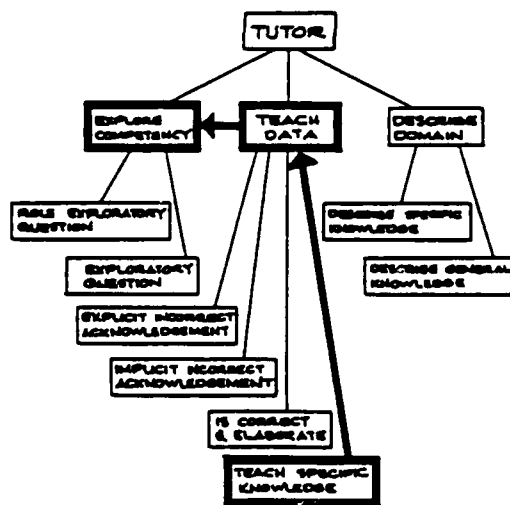
Figure 8: Informal Notation of Meta-rules.

The tutor begins in the state *explicit-incorrect-acknowledgement* (Snapshot 1, Figure 9). Being a tactical state, its principal action is to say something about a student's wrong answer, in this case "No." Having said this, the tutor still has "control" of the discourse and can continue to elaborate its response to the student's wrong answer. In the present design there is no default path out of *explicit-incorrect-acknowledge* at the tactical level. With a different set of rules, the tutor might, for example, continue speaking or it might reinforce the student's answer, perhaps by repeating it or elaborating part of it. However, since we

decided, in designing these rules, that the best thing to do at this point is to move to a higher planning level and to consider reformulating either the strategy or the pedagogy of the utterance, the tutor returns to the strategic level and to the parent state, *teach-data*, as indicated by the up arrow in Snapshot 1.



Snapshot 1

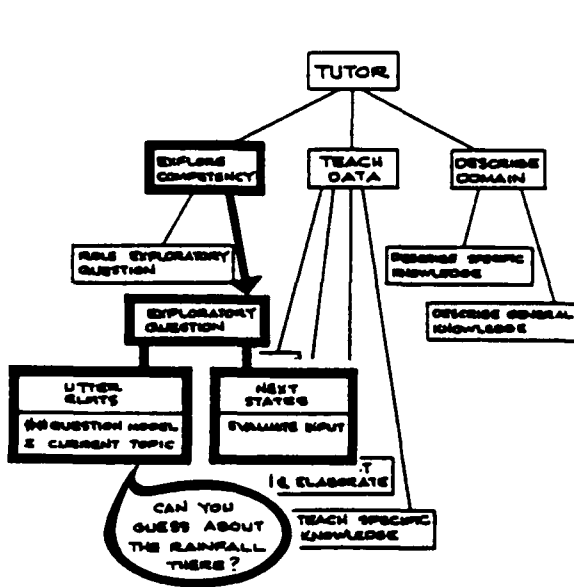


Snapshot 2

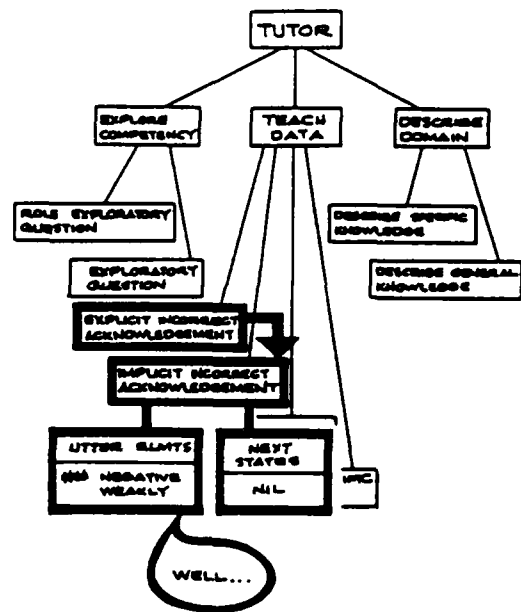
Figure 9: Snapshots of the DMN during Discourse Planning.

Once in *teach-data*, we take the default path down to the tactical level to *teach-specific-data*. In general at this point, a different meta-rule might have applied to take the tutor to a more particular tactical state, but in this case that did not happen. At *teach-specific-data*, as in any tactical state, the tutor says something and in this case it extends the utterance already begun with "No." The utterance is constructed from the specification built into this decision-unit and individualized by the values its elements have in this domain and at this point in the discourse. The specification is *specific-value* (*current-topic*), where *current-topic* has been carried forward from the previous ply of the discourse and is still "the climate in Washington and Oregon." The attribute value of this topic is "rather mild" (a canned phrase) and the surface language generator, MUMBLE [16], renders it in this discourse context (i.e., "full sentence") as "It's rather mild."

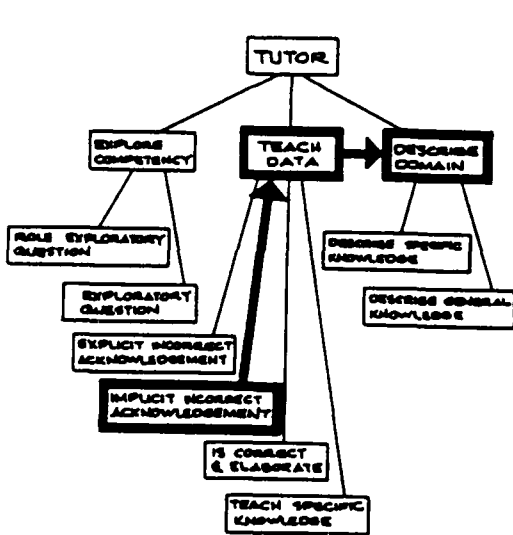
From *teach-specific-knowledge* there is no default path and the tutor moves up again to *teach-data* (Snapshot 2). This time, however, the context has changed and before *teach-data* can be evaluated, a meta-rule takes the tutor to a different decision-unit. The context has changed because the topics brought up until this point in the discourse have been answered or resolved. In detail, what happened was that, when the tutor supplied the correct answer to its own question (i.e., "It's rather mild"), the DMN register "question_complete" was set, satisfying one of the preconditions of the meta-rule, S1-EXPLORE (see Figure 8). The other precondition for this meta-rule was already satisfied, namely, that some topics related to the current topic remain to be discussed (as indicated by another register). When S1-EXPLORE is triggered it moves the tutor to *explore-competency*, in effect establishing that previous topics are complete and that a new topic can be explored. The next most salient topic in the knowledge base is "rainfall in Washington and Oregon" and it becomes the current topic.



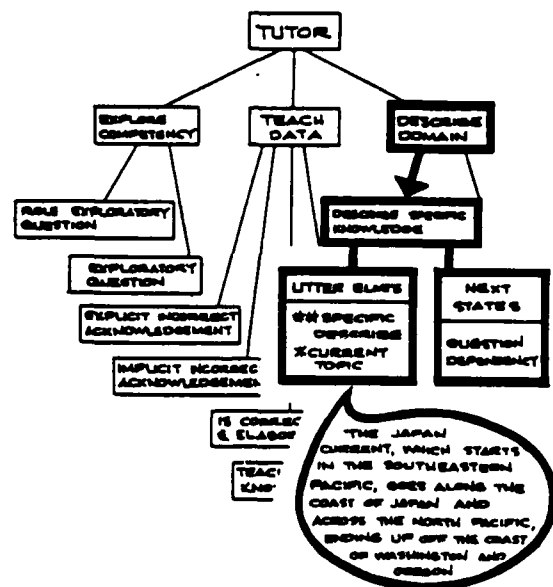
Snapshot 3



Snapshot 4



Snapshot 5



Snapshot 6

Figure 10: DMN during Discourse Planning (cont).

Once in *explore-competency*, the tutor takes a default path to the tactical level and to *exploratory-question* (Snapshot 3 in Figure 10) once again; and at the tactical level the tutor says something, in this case further questioning topics on the threshold of the student's knowledge. The utterance this time is constructed from the specification built into *exploratory-question*, which has been individualized by the values at this point in the discourse. The specification is *question-model (current-topic)*, where *current-topic* has been rebound to "rainfall in Washington and Oregon" at the time the meta-rule was enabled, as mentioned above. The utterance put out by Meno-tutor is "Can you guess about the rainfall there?"

At this point Meno-tutor moves to a default path and enters the tactical state *evaluate-input* which receives and evaluates the student's answer (not shown). His answer is wrong a second time and the default path moves the tutor, once again, to *explicit-incorrect-acknowledgement* where it would normally correct the student, as before. However, this state is not evaluated because the context is different and a new meta-rule, T6-A.IMPLICITLY, Figure 8, fires, moving the tutor to a new decision-unit (Snapshot 4). The context change is two-fold: 1) the student seems confused, and 2) the test for *wrong-answers-threshold* is met. Recognizing a confused student is admittedly a subjective and imprecise inference for a machine tutor. In this implementation, we have chosen to measure the student's confusion as a function of the number of questions asked, the number of incorrect responses given, and the extent to which the student's frontier of knowledge has been explored. In the example discourse, two questions were asked, two questions answered incorrectly, and the student's frontier of knowledge barely explored. Therefore, the student is judged to be confused and the meta-rule T6-A.IMPLICITLY triggered, forcing the system to move to the tactical state *implicit-incorrect-acknowledgement*. This move causes the tutor to

utter a refinement of its default response; instead of correcting the student, as did the default response of the previous utterance, text generated from this state implicitly recognizes, but does not dwell on, the incorrect answer and the tutor says "Well, . . ."

There is no default path from *implicit-incorrect-acknowledgement* and the tutor moves up to *teach-data* (Snapshot 5). Once again, a meta-rule takes the tutor to a new strategical decision unit, *describe-domain*. The context in this case is that the threshold of wrong answers has been met (as recorded by a register) and there exists at least one topic in the knowledge base ("Japan Current") which is linked to the major topic (the "climate in Washington and Oregon"). Based on the first fact, the system infers that the present strategy, *teach-data*, has been ineffective; based on the second fact, it infers that there remains an undiscussed geographical factor which, if described, could enable the student to infer the correct answer. S3-DESCRIBE is therefore triggered, moving the tutor to *describe-domain*. The action of this meta-rule terminates the interactive Q/A approach and begins a lengthy descriptive passage about the single topic, the "Japan Current."

From *describe-domain*, the tutor takes the default path to the tactical level and *describe-specific-knowledge* (Snapshot 6) and prepares to speak. The utterance specification in this state is *specific-describe (current-topic)*. As mentioned above, *current-topic* is now "Japan Current" and *specific-describe* has the effect of enunciating each attribute value of a specific topic in the knowledge base. Thus the description realized by Meno-tutor is "the Japan Current, which starts in the Southeast Pacific, goes along the coast of Japan and across the North Pacific, ending up off the coast of Washington and Oregon."

This brief tour through the DMN has illustrated the kind of knowledge that a tutoring system must have in order to make inferences about the content and approach of the tutoring discourse. It has also provided a view of the tutoring space available to Meno-tutor in that it can change the topics being tutored (rainfall vs. Pascal) along with the particulars of the tutoring rules for generating the discourse.

4. Pedagogical Annotation on Domain Knowledge

The second method used to structure tutoring discourse is to annotate the domain knowledge base with pedagogical information. Annotations "highlight" relevant topics and direct the planning of the text by keeping data per topic and per student, Figure 11. Topics are tagged in relation to a student's history and presumed level of knowledge; e.g., data that appears trivial in the context of current discourse can be skipped while salient data, especially if it has been "missed" by the student, can be explored in depth. Annotations play a role in tutoring comparable to that of size and centrality in photographs [17]: they identify parts of a knowledge base that are most relevant to the student and suggest strategies that will build on what the student already knows.

In this sense Meno-tutor incorporates a version of Goldstein's [18] Genetic Graph which he developed in the context of a computer coach. His genetic graph provided a structure to classify domain knowledge into skill categories being represented as knowledge a student would have used if he took correct actions in a computer game. Skills were linked to what the student was trying to learn and the topics he already knew while linguistic cues in the graph enabled the coach to generalize from, describe an analogy to, or suggest a deviation from, a skill that the student already knew.

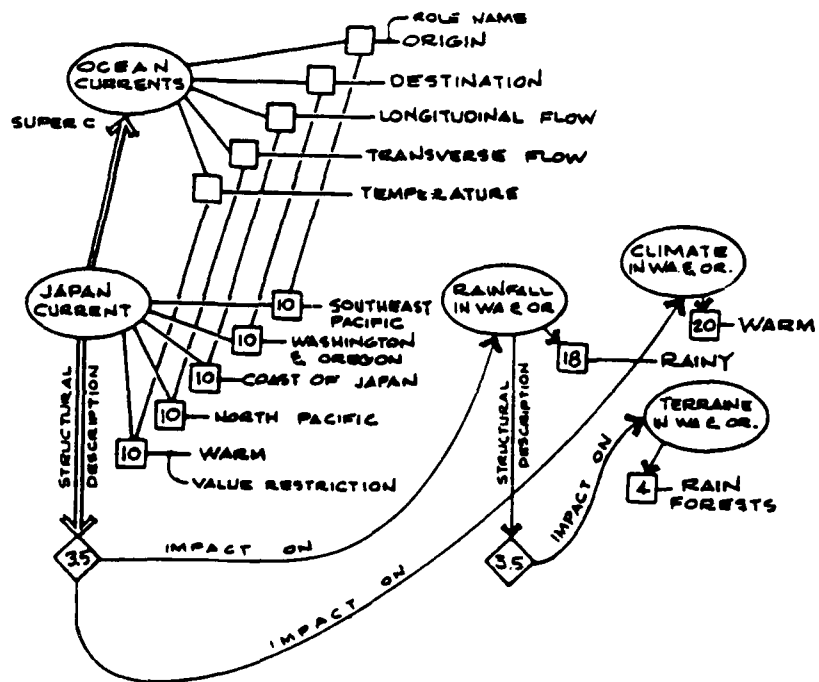


Figure 11: The Expert Knowledge Base for the Rainfall Discourse.

In Meno-tutor, annotations provide a relative focus of attention onto topics and relations that guide selection of teaching strategies; they do not, as in the case of the genetic graph, force a knowledge representation out of a limited number of explicit skills.

The annotations in Meno-tutor were developed from the work of Conklin [17] whose program planned paragraph-length descriptions of visual scenes under the guidance of an annotated knowledge base of the objects and spatial relations depicted in the scene. The annotations were derived from an empirically-based consideration of an object's visual *saliency* [19] and created a view of the domain "almost as if one could pick up the data base and shake it so that it dangled in order of decreasing saliency" [17, pp.]. Annotations served to highlight elements which, because of size, centrality, or unusualness in the scene would naturally be the first topics to be discussed.

Our present use of the annotated student model is similar to Conklin's in the sense that the model of the student's history is projected onto a general knowledge base, and the concepts or examples from which the student would profit are derived by matching his history to the contents of the domain.

Two complementary organizing schemes are used to adjust the basic planning mechanism and to define topics to be discussed, corrected, or questioned. They provide planning templates that act as guideposts for searches through the knowledge. The first annotation scheme, Risland's *Michelin Rating* (MR) [20], encodes an intrinsic importance value to each topic without regard for particular student. It provides a way to quickly retrieve topics ordered by teaching or learning priority. This numeric annotation provides a coarse way to pull topics out of the knowledge base ordered by importance. The numbers were chosen by the author, based on empirical studies of learning and teaching in the subject area. The Michelin Rating is comparable to the importance number used by Collins et al. [21].

The second annotation scheme, the *Expected Competency* (EC), acts as a focus of attention for tutoring and as such represents the system's view of the student's knowledge item by item. At the beginning of the session, the EC value is set without regard for the student's competency; it is dynamically updated during discourse as a record of the tutor's confidence in its own evaluation of the student's knowledge. A raised EC indicates an increase in knowledge of the topic; a lowered EC value indicates a wrong response or decreasing evidence of knowledge of the topic.

The EC value is updated at each ply of the discourse and checked for support for, or deviations from, the given value. For instance, if the EC had a high value for a particular topic, it implied that the student had learned the topic and, presumably, all topics of lower MR value. However, if the student subsequently made errors on this or lower topics, it would suggest that, in fact, the student did not know topics at this level. In such a case, the tutor would adjust its EC value downward to be consistent with an apparent lower threshold of knowledge. On the other hand, if the EC was accurate, additional correct answers on this topic, and topics of lower MR value, would support the system's confidence in its EC value.

The EC and MR act together as a focus of attention to determine the type of actions to be generated by the tutor. For instance, if annotations suggest that the topic is salient (high MR) and the student's knowledge negligible (low EC), additional explanations and examples will be favored in the dialogue as opposed to additional questions. Alternatively, if the annotations had a high EC value, indicating some level of student knowledge, the tutor will reduce its interactions with the student.

Additionally, annotations are used to regulate the amount of interaction between tutor and student. For example, a succession of correct answers confirms an existent high EC value, and ultimately leads to shorter responses from the tutor, to the point where a single student error on the heels of several correct answers would be overlooked as a momentary lapse. On the other hand, a lower EC raises the system's attention to a student's answer and moves to more frequent intervention and correction.

The combined annotation scheme allows the system to avoid inappropriate topics, including those *below* the student's learning threshold (they would be too easy), and also those *above* it (they would be too hard). In this sense, Meno-tutor implements the genetic graph by teasing apart discourse strategies and domain knowledge and teaching outward from the student's presumed threshold of knowledge. Unlike the genetic graph, Meno-tutor does not "rewrite" domain knowledge into skills or constrain the system to express its information about the student's actions as a specialization or generalization of an earlier used skill.¹ Rather, Meno-tutor enables domain knowledge to be expressed in a variety of ways and allows multiple pedagogical information, e.g., specializations and examples, to be added to that representation as a way of providing flexibility and expressiveness to the system.

We explored the power of this system of pedagogical annotations in two ways. In addition to producing alternative tutoring discourses by changing the domain knowledge base and annotations, we explored the effect of changing the annotations on a constant domain knowledge base. We wanted to demonstrate the power that ordering domain topics had on the production of discourse and we suspected that a knowledge base annotated with different

¹ The genetic graph existed as a module apart from domain knowledge, thereby duplicating information stored there. Alternatively, the skills designated in the genetic graph could have been implemented as the only expression of the domain knowledge base, thereby limiting the system's knowledge and expressiveness.

parameters would provide a very different "view" of the topics to be tutored.

We changed the MR on the rainfall domain knowledge base, which had the effect of imposing a distinct importance hierarchy onto the list of topics to be discussed. Figure 12 illustrates the new ordering on the original rainfall data base and Figure 13 shows the proposed dialogue. The discourse was not actually generated by Meno-tutor, but the changes appear to be straightforward. Since the new Michelin Ratings highlight the concept of

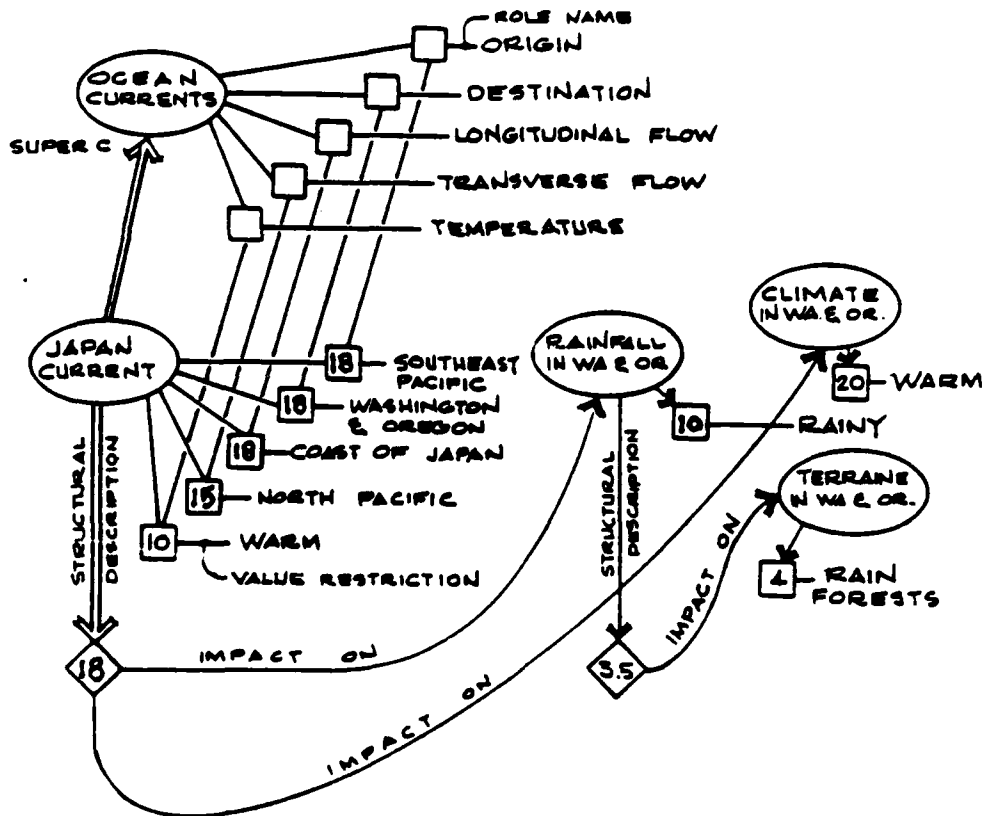


Figure 12: The Rainfall Knowledge Base with New Michelin Rating.

"Japan Current" over the concepts of "warm climate" or "heavy rainfall," the roles of the "Japan Current" are given priority over the general descriptors. As the student continued to give correct answers to the questions about the Japan Current, new role topics were selected for questioning.

TUTOR: Do you know what the climate is like in Washington and Oregon?

STUDENT: Is it cold?

TUTOR: No, it is rather mild. Can you guess about the effect of the Japan Current on the climate there?.

STUDENT: I don't know.

TUTOR: Well, the Japan Current begins in south eastern Pacific. Can you guess in which direction it travels?

STUDENT: Northwardly?

TUTOR: That is right. Do you know where it ends up?

STUDENT: Near Washington and Oregon?

TUTOR: Good. How do you think that current affects the current in Washington and Oregon?

STUDENT: It's probably rainy.

TUTOR: It is indeed. There are rain forests in both Washington and Oregon. What does that tell you about the temperature of the Japan current?

STUDENT: It's warm.

Figure 13: Discourse Produced by Varying Michella Rating.

The biggest problem with building knowledge bases for tutoring systems is how to organize and order domain topics. Such an organization must include relations between topics, topic successors, topic predecessors, preknowledge, examples, analogies, and topic rankings. Without such knowledge, design of machine discourse remains in the hands of the system author and one author's selection of salient topics might be another author's selection of trivia.

Since few empirical studies treat learning and teaching in specific domains, (see for example mathematics [20], chess [22], programming [23, 24], and physics [25, 26]), the optimal organization of topics in most domains is unknown. Even empirical studies, when used in conjunction with a tutoring system, are barely adequate to completely define the organization and ordering of the knowledge base. Expert tutoring systems require the coevolution of empirical studies of learning and teaching in each domain; without such research, which is scarce today, development of intelligent tutoring systems reflects the idiosyncratic view of the particular system designer.

In addition to studying the effect of modifying the annotations on the domain knowledge base, our second exploration of the effect of pedagogical annotation was to substitute a new knowledge base for the facts about rainfall and to use a knowledge base about Pascal looping concepts that drew on extensive cognitive studies about how novices learn Pascal constructs [14, 27]. The Michelin Ratings for the Pascal knowledge base were supplied by Woolf based on teaching experience and rich cognitive studies. These cognitive studies, and Meno-tutor itself, were part of a larger research effort directed at building an on-line, run-time support system for novice Pascal users [13, 14]. As a part of this effort, a Bug Finder [15] was also developed and used for four semesters on classes of several hundred students. The Bug Finder detected run-time semantic errors in Pascal programs and

passed messages on to the tutor about the location of the error, the names of variables associated with the error, etc. The Bug Finder has since been retired and the discourse in Figure 8 was generated based on simulated messages from the Bug Finder and simulated student input.

The program in Figure 14 was submitted by a novice programmer and the dialogue it engendered from Meno-tutor is reproduced in Figure 15. Given the program of Figure 14 and the parts of the WHILE loop which were programmed correctly, the tutor inferred that the student did indeed understand the basics of loop programming. To be certain that the tutor and the student shared a common vocabulary, the tutor asked two questions; both were answered correctly, suggesting that the student understood the rudiments of looping constructs and the role of the control variable. In the third question, the Meno-tutor analyzed the student's grasp of deeper programming concepts. Prior studies [23] had linked several explicit

```
1  PROGRAM LESSON1(INPUT, OUTPUT);
2  VAR
3  SUM, GRADES, STUDENTS: INTEGER;
4  MEDIAN: REAL;
5  BEGIN
6  SUM:=0;
7  STUDENTS:=0;
8  READ(GRADES);
9  WHILE GRADES>0 DO
10 BEGIN
11 SUM:=SUM + GRADES;
12 STUDENTS:=STUDENTS+1;
13 GRADES:=GRADES+1;    <=== should be READ (GRADES);
14 END;
15 MEDIAN:=SUM/STUDENTS;
16 WRITELN
17 ('THE MEDIAN GRADE IS', MEDIAN:3)
18 END.
```

Figure 14: A Student Pascal Program.

programming errors with conceptual misunderstandings of looping constructs. For instance, the programming error in this student's program has been linked with a belief that (1) a single call to READ will access all the values of a variable at once, or that (2) updating the control variable (line 13) will do a tacit READ through each value of the variable each time through the loop. The tutor's third and fourth questions discriminate between these two misconceptions and indicate, as is often the case, that the student had both misconceptions (both answers were wrong). The tutor's last response is an example of the "grain of truth" speech pattern: it confirms and reinforces the student's correct knowledge about programming while pointing out the inappropriate components of that answer.

A portion of the Pascal knowledge base is represented in Figure 16. It was encoded in the same kind of KI-ONE semantic network as was the rainfall discourse. Thus the Pascal knowledge base was available for interpretation by the same techniques as used in the earlier discourse. However, this knowledge base was richer than the rainfall knowledge base because it contained assumptions about incorrect, yet commonly held, programming misconceptions.

In sum, the annotation process provides a heuristic for ordering and organizing discourse topics and helps to individualize the dialogue for an individual student. The hooks into the expert knowledge source, in the form of tutoring information, act as pointers to text generation strategies. In this way pedagogical as well as rhetorical alternatives can be tested and identified while the system is in use. These conversational alternatives were used during development of the third component of the system, representation of discourse conventions, and tutoring maxims, as described in the next section.

Discourse Produced Using a PASCAL Data Base

Tutor: Do you know that GRADES, in line 9, is a control variable for the WHILE loop?

Student: Yes

Tutor: Does that mean that GRADES should be
a) Initiated before the loop?
b) Added into SUM within the loop?
c) Modified within the loop?
d) Both a) and c)

Student: C

Tutor: You seem to understand about the control variable of the WHILE loop. Now let's talk about some other command statements. What do you think READ (GRADES) in line 8 does with the value of GRADES?
a) Reads all values of GRADES at one time.
b) Reads in a single value of GRADES.
c) Assigns several new input values to GRADES.

Student: A

Tutor: Then do you think that GRADES := GRADES + 1 updates the next of those values?

Student: Yes

Tutor: There is your misconception. Adding one to a variable is a common way to step through a range of values. But it cannot apply here because your READ instruction in line 8 retrieves only a single value. You should replace line 13 with a second call to READ (GRADES).

Figure 15: Tutoring for Errors Made in the Program of Figure 14.

5. Qualitative Reasoning about Tutoring Discourse

The third method used to organize tutoring discourse is to encode knowledge of discourse conventions into the discourse decision making process. Many of the discourse decisions described above for the DMN and the annotated knowledge base were predefined by the authors, e.g., meta-rules governing discourse moves and algorithms used to evaluate a student's knowledge were implemented according to the authors' own view of teaching and

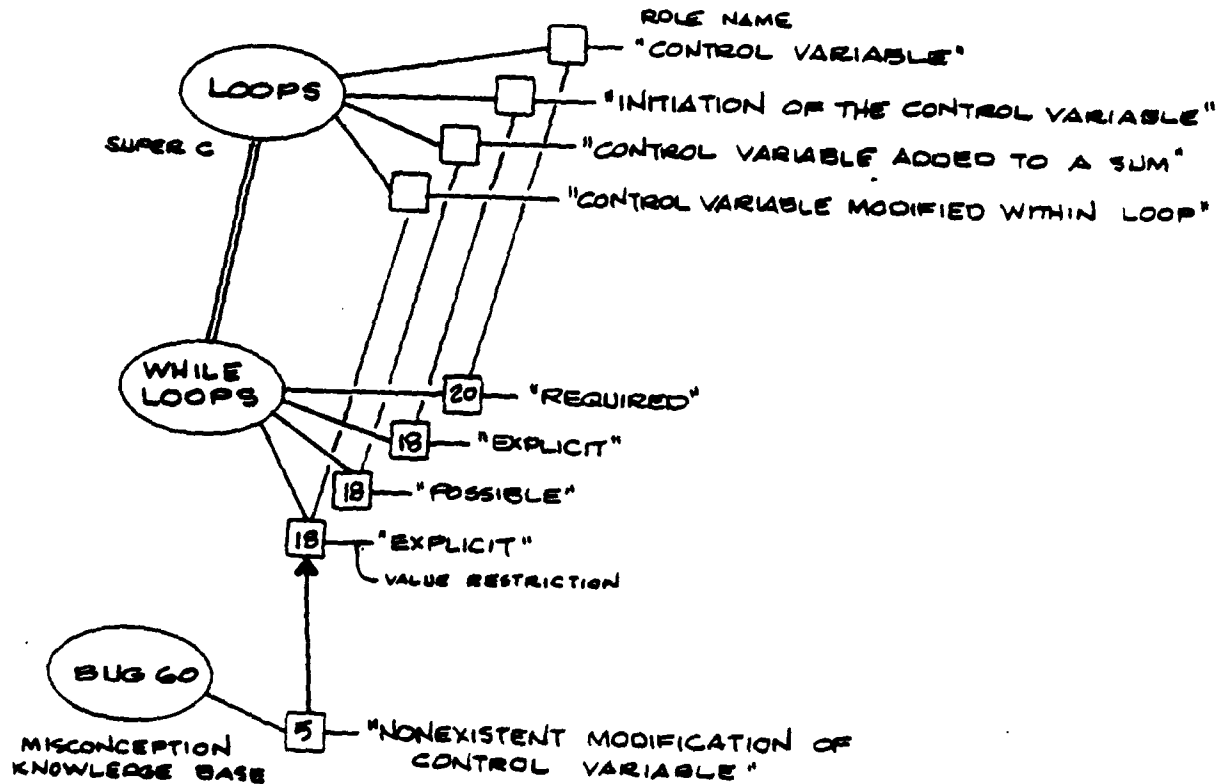


Figure 16: The Expert Knowledge Base for the Pascal Discourse.

learning in the two domains. This limited first step towards developing a machine tutor does not enable the machine to reason about how and when to shift discourse or how to evaluate student actions. In this section we describe a theoretical view of human tutoring and discourse designed to remedy this situation. We suggest a representation for inferences and expectations made by humans in discourse. We have two goals in mind in defining such a representation: 1) to use expectations and inferences to motivate discourse decisions and 2) to identify a set of maxims that underly effective tutoring.

5.1 Expectations in Discourse

The first goal in representing human expectations in discourse is to use qualitative assessments made about discourse to govern the mechanical tutor's response options. Human speakers and listeners have expectations about each other and about the underlying structure of the discourse in which they are engaged. For example, they use subtle linguistic cues to relate their current utterance to preceding ones, to shift topics, or to provide supplementary knowledge to each other. These expectations set up by listeners are what the human speaker tries to anticipate and to deliberately control for.

That qualitative judgements about the appropriate affect of discourse do exist should not be controversial and can be illustrated by an example. Consider the range of affect that might be used in a query about loop execution in a Pascal program shown in Figure 17. Each sentence has a similar locutionary force, yet each conveys a different attitude toward the student on the part of the tutor. Further, there is a continuum such that a tutor may couch his statements at any place along the *top* of list of responses and the implied affect would be one of close attention, even commitment, to the student. On the other hand, a statement selected from the *bottom* of the list would imply non-commitment, non-involvement, and possibly antagonism. Relative to the four utterances, we say that use of a phrase representing a certain point on the scale carries the implication that the tutor chose not to

-
- If the input is 10, how many times would your loop execute?
 - Do you know how many times your loop would execute?
 - I bet you don't know how many times your loop will iterate.
 - You couldn't possibly understand loop execution.

Figure 17: Implications of Utterances.

phrase the utterance by other expressions lower on the list. This reasoning on the part of the listener is licensed by the Gricean [28] maxim of manner, which, along with his very general maxims for discourse, are evocative of discourse, though not yet a detailed basis from which we can fashion a complete computational theory of discourse.

Our computational model of discourse includes qualitative assessments and constraints about discourse that are used to govern the mechanical tutor's options. For example, if the machine interprets a discourse move such as makes accusation, as in Figure 17, it will record the negative connotations associated with the utterance. Similarly, any choice of discourse move causes specific inferences to be made about the speaker's intention (or knowledge) in addition to those inferences made about his actual words.

In our computational model of discourse, inferences about the speaker's intent triggers meta-rules that move the tutor into a new state sequences, Section 3. The process is analogous to the read-eval-print cycle of LISP where the top-level "thinking" of the machine is suggested in Figure 18. The first step is to interpret a student's response which in turn triggers mechanical inferences that are placed in a heap. Those inferences that are supported

-
- STEP1: Behave according to default state sequence (consistent with current inferences).**
A) generate text
B) interpret student's response
- STEP2: Identify inferences of student's literal utterance and endorses evidence**
- STEP3: If endorsed inferences reach threshold, trigger meta-rules, and move system to a new state sequence.**
- STEP4: Go to 1.**

Figure 18: Steps of the Intelligent Tutor.

[Sullivan and Cohen] are endorsed and given reasons to be believed or disbelieved. Endorsements are associated with an applicability condition, e.g., "correct answer indicates correct information" is always possible when the response is correct; "correct answer indicates a guess" is applicable when the response is correct but earlier responses were wrong; "could be a mistake" is applicable for any response. Inferences that pass beyond threshold will be used to activate changes in the system's behavior. Some endorsements are *negative*, meaning they provide reasons to disbelieve their associated interpretations. Others are *positive*, meaning they support the interpretation with which they are associated.

The state of affairs of a discourse is represented by the set of assessments that are either positive or negative support of prior statements. When evidence for a change in the current interpretation or "belief" for a new implication is presented, the system will take action by activating a meta-rule which in turn changes the teaching or discourse response.

5.2 A Computational Model of Implicatures and Assessments

To make inferences about discourse moves, we began with Grice's original formulation of implicatures [28]. He suggested that implicatures resided within certain words, for example, the italicized words in Figure 19: The word *and* in the first sentence carries an implication that the activity of going to jail preceded, and possibly caused the second activity, that George became a criminal. The use of the word *tried* in the second sentence carries with it an entailment that Millie failed to swim the English channel, and the use of the phrase *one*

- 1) George went to jail *and* became a criminal.
- 2) Millie *tried* to swim the English channel.
- 3) I have *one* leg.

Figure 19: Implicatures in text.

leg in the third sentence, implies that the speaker *does not* in fact have two legs. Implicatures include the desiderata normally accepted by a rational discourses and embody a speaker's motivation, intention, and involvement in the discourse.

In our computational model of discourse, *implicatures* are represented as bound to discourse moves. Implicatures exist independent of the "truth" or "meaning" of the utterance and define what the listener receives in addition to the spoken words. This qualitative implicature is placed on a heap whenever its move-class is invoked. Figure 20 lists the implications bound to two move-classes, question topic and correct-answer. For instance, if a student questions the tutor about a topic, the implications of this are that the student 1) thinks the topic is important, 2) knows (or is trying to learn) topics on his threshold of knowledge, 3) thinks the topic is learnable through discourse. These implications will be assumed by a listener independent of the content of the query.

Typical objects in our ontology

(define-move-class QUESTION-TOPIC

Evidence:

- Q+ *topic is important*
- Q+ *topic is within threshold of knowledge*
- Q+ *topic is learnable through discourse)*

(define-move-class CORRECT-ANSWER

Evidence:

- Q+ *topic is generally known*
- Q+ *topic is background information*
- Q++ *answer is a guess)*

Figure 20: Implications bound to move-classes.

- *Student-has-definitional-knowledge - student has marginal knowledge of the topic, perhaps derived from knowing its definition.
 - *Student-has-background-information - student displays prior experience with topic.
 - *Student-is-confused - student displays contradictory information.
 - *Student-understands - student appears knowledgeable about this topic.
 - *Student/domain-agreement - there is some agreement between student's information and domain knowledge.
 - *Student-knowledge-threshold-known - We have identified both known and unknown topics in the domain.
-

- *Topic-is-important - topic is relevant in this domain.
- *Topic-is-generally-known - several components of the topic appear well understood.
- *Topic-is-learnable-elsewhere - topic was learned before the discourse.
- *Topic-is-on-student-threshold - topic is closely related to both known and unknown topics.
- *Topic-is-complete - topic was fully developed during discourse.

Figure 21: Global Assessments Made by the System.

Global assessments are based on extended reasoning over sequences of implicatures. They include inferences such as *student is confused, *topic is known or *misconception is resolved and are modified with each new tutor/student interactions. Global assessments are heuristic and represent the system's best estimate about the state of affairs of knowledge of the student over a sequence of discourse moves. Whereas implicatures are always possible, tempered by aggregated inferences from prior utterances, global assessments can never be assumed, but require reasoning under uncertainty to deduce which one of a number of competing global assessments might take effect. A number of global assessments are presented in Figure 21. The assessment is listed on the left and the inference of which the assessment is a "gloss" is on the right. Reasoning with uncertainty allows us to accumulate support for or against a number of these assessments.

5.3 Using Inferences in Discourse Decisions

Tutor: Do you know that *GRADE* in line 8 is a control variable?

Student: Yes

```
; IMPLICATURES
; *student_has_definitional_knowledge
; (*topic control_variable) is_generally_known
; (*topic control_variable) is_learnable_elsewhere
; (*topic control_variable) is_background_material
; response_was_a_guess
```

Tutor: Good. What is the value of grade before leaving the loop in line 13?

Student: 9999

```
; IMPLICATURES
; *student_has_definitional_knowledge
; (*topic value_control_variable) is_generally_known
; (*topic value_control_variable) is_learnable_elsewhere
; (*topic value_control_variable) is_background_material
; response_was_a_guess
```

```
; GLOBAL ASSESSMENTS
; *student/domain_agreement
; *student_knows_the_topic
```

Tutor: That's right. What is the value of grade after leaving the WHILE loop, in line 13?

Student: I don't know.

```
; IMPLICATURES
; *tell_tale_signs_lack_of_knowledge
; *student_does_not_know_topic

; GLOBAL ASSESSMENTS
; (*topic value_control_variable) is_on_student's_threshold
; *student/domain_disagreement
; *student_is_confused
```

Figure 22: Using Inferences to Motivate Discourse.

As an example of the use of implicatures and assessments in making discourse decisions, we present three questions that might be asked of a student who had submitted an incorrect Pascal program, see Figure 22. The first correct response places immediate implicatures on the heap: The topic is generally known or learnable through other efforts (i.e., textbooks or lectures), the topic was studied as background material or the student's answer was a guess. After two correct answers, the tutor can reinforce its initial evaluation of the student's knowledge but now is licenced to make more extensive inferences about the student or the topic. In this case, the global assessment is that there is some agreement between the student's information and the domain knowledge base. Such an inference is possible because evidence from a second correct answer provided support for the global assessment.

The student's third response is wrong and the tutor is now forced to reverse its current evaluation. After a single wrong answer, several immediate implicatures are available since they are bound to the conversational move: either the student does not know the material in question or he made a careless error. If we assume the former and recognize that the wrong answer came on the heels of two correct answers, we have a more complex assessment possible: the topic might lie on the student's threshold of knowledge. This assessment is licenced by the fact that the student correctly identified an example control variable, perhaps by using its definition, yet he incorrectly indentified its value after loop exit.

The example shows how the tutor can make general assessments about the student over several interactions using evidence brought from earlier responses and weighed along with current implicatures to generate a more global view. In this way the tutor can achieve a broader view of student knowledge and topic complexity.

The three questions presented above come from our Pascal tutor (See Section 2) and show the kind of power we want from a discourse inferencing mechanism. The next two figures complete this example and Figure 23 shows the student received the same Pascal problem as did the student of Figure 14. However the second student produced a different program with a new set of semantic errors. Again this program is syntactically correct but produces run time errors.¹ It reveals at least four underlying misconceptions about control

PROBLEM: Write a program that finds the average grade for a student who types his grades in at the keyboard. After the last grade is typed in the student will type 9999. Please print out the average grade.

```
1 Program Student29 (input, output);
2 Var
3   sum, num, grade, ave : integer;
4 Begin
5   sum := 0;
6   num := 1;
7   read (grade);
8   while grade <> 9999 do
9     begin
10    read (grade);
11    sum := sum + grade;
12    num := num + 1
13    end;
14   while grade = 9999 do
15     begin
16    ave := sum/num;
17    writeln (ave)
18    end;
19 End.
```

Figure 23: A Student Program.

¹ • The program is flawed in several ways. The value of the "read" command in line 7 will not be processed, because the "read" command in line 10 immediately replaces it with a new value. The second loop will never terminate because the control variable "grade" is never modified inside the loop body. We also suggest, that the student assumes that the second loop will execute only once since its only action is to print out the average. We call such a loop an "if-loop" and suggest that the student has confused the "if" conditional construct with the family of looping constructs.

variables, looping constructs, and flow of control. Cognitive studies of programming [14, 16, 17] suggest for instance, that the student may believe:

>> the order of operations inside the loop should be READ/PROCESS (a Pascal WHILE loop is designed to be PROCESS/READ);

>> the value of a variable inside the loop is distinct from its value outside the loop;

>> the loop construct does not test the last value of the control variable; and

>> a single iteration of a sequence of commands requires a looping construct.

There is evidence for each misconception.⁹ For instance, the fact that "read (grade)" in line 10 precedes the processing of grade in line 11 is evidence for the first misconception. In addition, the use of two loops, both controlled by the same variable, "grade," is evidence for the second and third misconceptions since they suggest that the student does not realize that the value of "grade" will be available at the end of the first loop and therefore does not have to be retested. In the example, the tutor's goal is to determine the breadth of the student's understanding and to resolve the ambiguity among the possible misconceptions. The object is to focus the discourse on the repair of a single misconception. The discourse suggests how our tutor might unravel the student's misunderstandings and make him "verbalize" his malformed ideas. In order to generate that discourse, the tutor must:

focus on several topics in succession (e.g., control variables and while loops);

handle several errors simultaneously (e.g., value the control variable and the number of loop iterations); and

customize new examples to present to the student (e.g., a hypothetical program and a new problem).

1. *Tutor:* Do you know that *GRADE* in line 8 is a control variable?
Student: Yes
2. *Tutor:* O.K. What is the value of *GRADE* at anytime during loop execution?
Student: Depends on the value read in.
3. *Tutor:* Right, and what is the value of *GRADES* immediately after leaving the *WHILE* loop, in line 13?
Student: I don't know.
4. *Tutor:* O.K., now suppose the values read in were 1 5 9 10 9999.
How many times would your *WHILE* loop execute.
Student: I don't know. Maybe 5 times.
5. *Tutor:* Well, what would be the value of *GRADE* immediately after leaving the loop in line 13?
Student: I don't know.
6. *Tutor:* You seem to have a misconception. The loop would execute 4 times. Although 9999 would be read inside the loop during the fourth iteration and tested at the top of the fifth iteration, the entrance test for that iteration would fail since *GRADE* is now unequal to 9999. The fifth loop execution would never occur and control would pass to line 14 immediately after the loop.

You probably assumed that the value of the variable inside the loop was different from its value outside the loop. In fact, whether inside or outside of the loop, variables in a *WHILE* loop have the same value. If *GRADE* equals 9999 before the loop terminates, it will still equal 9999 after the loop terminates. Testing the value of *GRADE* immediately after the *WHILE* loop is superfluous.

Now, let me give you a new problem: Compute and print the average number of hours worked each day by a student employed during a month if hours per day is typed in and averaged by the program. Assume that hours per day are typed on a single line and followed followed by -1.

Figure 24: Proposed tutoring discourse for the Program in Figure 9.

Note that in Figure 24 the tutor asks one question (line 1) to establish that both it and the student share a common vocabulary about control variables. In the next two questions (lines 2-3) the tutor asks enough questions concerning misconceptions about variable values and control flow to establish that the student does, in fact believe that the value of

grades is not available after the first loop terminates. In line 4 the tutor presents example input that has been custom-tailored to the student's knowledge of the problem. Its goal is to verify the hypothesis that the student did not realize that the value of *grades* was available after the loop exited. Based on the student's response thus far the tutor (line 6) explains its diagnosis of the misconception in terms of characteristics of the presenting program.

5.4 Tutoring Maxims

In the previous 3 sections we used inferences about discourse expectations to govern the mechanical tutor's choice of response options. In this section we describe the second goal behind the representation of discourse conventions, namely to identify maxims that underly effective tutoring. Based on a number of studies about tutoring and discourse strategies [11, 28, 29, 30, 31], we have begun to identify maxims that allow good human tutors to produce effective discourse. Some of these discourse descriptors include:

Quality: be committed to the student and interested in him;
support the student; be co-operative with him;

Quantity: be specific and perspicuous;
use a minimum of attributes to describe a known concept;

Relation: be relevant;
find a student's threshold of knowledge;
bring up new topics and viewpoints as appropriate to the student's threshold

Manner: be in control;
organize the process of moving from topic to topic, but
allow the student to take some initiative and
allow context to determine a new topic;
return to complete unfinished topics;

Maxims

Be co-operative:
-work with student

Be committed:
-show interest

-support student

Be relevant:
-find student's threshold

-teach at threshold

Be organized:
-structure domain

-complete information

Be in control:
-strictly guide discourse

**Conversational
move-classes**

explain topic
summarize topic
clearly terminate topic
review or repeat topic
release control of dialogue

acknowledge answer
explain topic

outline topic
introduce topic

question student
evaluate student hypotheses
propose and verify misconceptions

provide analogy example
summarize topic

outline topic
introduce topic
terminate topic
review topic

clearly terminate topic
teach subtopic after topic
teach attributes after topic
teach subgoal after goal

introduce topic
describe topic
question student

Figure 25: Tutoring Maxims supported by move-classes.

Figure 25 proposes a further way to discriminate these maxims in terms of discourse moves that support each one. Obviously, a variety of discourse moves could have been selected to implement each maxim. The maxims we have chosen are listed on the left and the sequence of moves supporting them on the right.

The linkage between maxims and support moves can be read in two ways: from *left to right* it offers a way to plan discourse and from *right to left* it provides a way to evaluate the effect of the discourse on the student. In the first direction a maxim on the left is used to plan the discourse and the utterance to be generated "shaped" to fit into the move-class on the right. For instance, being maximally *organized* is often a top goal for a tutoring system. If this is the case, independent of the particular content of the next response, the system will try to couch its next utterance in terms a discourse move associated with the organizational maxim: *outline topic, introduce topic, terminate topic, or review topic*. The system can tolerate movement away from this particular goal, but if the goal is a top one, the discourse algorithm must ultimately return language generation to the specified discourse moves.

In the second direction, tutoring moves on the right are used to compile a record of the discourse interaction. By "abstracting" from moves, on the right, to maxims that subsumes it on the left, the system can have some reading of the presumed "effect" of its discourse on the student. For instance, if the interaction with the student has included moves such as, *explain topic, clearly terminate topic, and release control of discourse*, the machine can judge that the overall effect of the discourse might be seen as being co-operative. (Obviously, a few make accusations within the discourse could negate any such impression.)

In sum, we have used implicit inferences and expectations in discourse to recognize qualitative states, such as *topic is generally known, *student has background information, or *student is confused. Discourse moves are represented in terms of implied expectations elicited in the student. Responses by the student, categorized, for instance, as correct-answer in the context of a number of prior partially-correct-answers, is similarly taken to support expectations by the tutor such as *student-understands or *student-is-confused. Changes in any of these states may trigger meta-rules that change the tutor's response.

We make associations between discourse moves (as instances of move-classes) and maxims in terms of qualitative evidence. Each discourse move is defined as a data structure with two associated inferences: implicatures and global assessments. Implicatures are linked directly to a discourse move and represent an inference made about the move itself; they are fixed and non-negotiable. Global assessments are linked indirectly to sequences of move-classes and represent inferences made about an effect over several move-classes; they are volatile over the life of the dialogue. Implicatures and global assessments are used in conjunction with the discourse manager (Section 3) to move the system from one set of discourse states to another.

6. Summary

We have suggested that because tutoring is a process of specialized communication, the tutor's ability to refine its discourse to the domain and to the student is central to its success. Since tutoring requires anticipating a student's unspoken inferences and expectations, a system tutor must have a way to recognize a student's misconceptions

and to reason about discourse elements that are qualitative, rather than quantitative, that reflect intentions rather than speech acts.

We have described the control and data structures of Meno-tutor as a way of showing how a tutoring system using Artificial Intelligence techniques can plan and generate its discourse. We have described three computational elements designed to customize a machine's response to an individual student: a discourse management structure, annotation of the student model, and qualitative inferences about the student and topic. In each of these mechanisms, we have looked at ways to dynamically record and update information about the student or topic based on knowledge of discourse. Our work focuses on the deep level planning required to motivate discourse rather than on the generation of syntactically correct natural language output.

7. References

- [1] Woolf, B., *Context-Dependent Planning in a Machine Tutor*, Ph.D. Dissertation, Computer and Information Sciences, University of Massachusetts, Amherst, MA, 1984.
- [2] Woolf, B., and McDonald, D., "Design issues in building a computer tutor," in *IEEE Computer*, special issue on "Artificial Intelligence for Human-Machine Interaction," Sept. 1984.
- [3] Stevens, A., Collins, A., and Goldin, S., "Diagnosing student's misconceptions in causal models," in *International Journal of Man-Machine Studies*, 11, 1978 and in Sleeman & Brown (eds.), *Intelligent Tutoring Systems*, Academic Press, Cambridge, MA, 1982.
- [4] Allen, J., *A Plan-Based Approach to Speech Act Recognition*, in Brady & Berwick (eds.), *Computational Models of Discourse*, MIT Press, Cambridge, MA, 1983.
- [5] Brown, J. S., Burton, R., and Bell, A., "SOPHIE: A sophisticated instructional environment for teaching electronic troubleshooting (An example of A.I. in C.A.I.)," *International Journal of Man-Machine Studies*, 7, 1977.
- [6] Burton, R., and Brown, J. S., "An investigation of computer coaching for informal

- learning activities," in *International Journal of Man-Machine Studies*, 11, 1978, also in Sleeman & Brown (eds.), *Intelligent Tutoring Systems*, Academic Press, Cambridge, MA, 1982.
- [7] Mann, W., Moore, J., and Levin, J., "A comprehension model for human dialogue," *International Joint Conference on Artificial Intelligence*, 1977.
- [8] McKeown, K., "Generating relevant explanations: Natural language responses to questions about data base structure," *National Proceedings of the Association of Artificial Intelligence*, 1980.
- [9] Finin, T. W., "Providing help and advice in task oriented systems," *Proceedings IJCAI-83*, Karlsruhe, W. Germany, 1983.
- [10] Wilensky, R., "Talking to UNIX in english: An overview of UC," *Proceedings AAAI-82*, Pittsburgh, PA, August 1982.
- [11] Clancey, W., "Tutoring rules for guiding a case method dialogue," *International Journal of Man-Machine Studies*, 11, 1978, also in Sleeman & Brown (eds.), *Intelligent Tutoring Systems*, Academic Press, Cambridge, MA, 1982.
- [12] McDonald, D., "Natural Language Generation as a Computational Problem: an Introduction," in Brady & Berwick (eds.), *Computational Models of Discourse*, MIT Press, Cambridge, MA, 1983.
- [13] Soloway, E., Woolf, B., Barth, P., and Rubin, E., "MENO-II: An intelligent tutoring system for novice programmers," in the *Seventh International Joint Conference on Artificial Intelligence*, Vancouver, Canada, 1981.
- [14] Soloway, E., Bonar, J., Woolf, B., Barth, P., Rubin, E., and Erlich, K., "Cognition and programming: Why your students write those crazy programs," *Proceedings of the National Educational Computing Conference*, NECC, No. Denton, TX, 1981.
- [15] Rubin, E., *A Bug Finder for Pascal Programs*, Unpublished Masters Thesis, University of Massachusetts, Amherst, MA, 1981.
- [16] Bonar, J., "Collecting and analyzing on-line protocols from novice programmers," *Behavioral Research Methods and Instrumentation*, 1982.
- [17] Conklin, E., "...." Ph.D. Thesis, University of Massachusetts, Amherst, MA, 1983, also available as Tech Report #:#.
- [18] Goldstein, I., "The Genetic Graph," in Sleeman & Brown (eds.), *Intelligent Tutoring Systems*, Academic Press, Cambridge, MA, 1982.
- [19] Conklin, E., Erlich, ., and McDonald, D., "Visual Saliency," *Cognitive Science Journal*, 1984.

- [20] Risland, E., "Understanding understanding mathematics," *Cognitive Science*, Vol. 2, No. 4, 1978.
- [21] Collins, A., Warnock, E., and Passafiume, J., "Analysis and synthesis of tutorial dialogues," *Psychology of Learning and Motivation*, vol. 9, Academic Press, Inc., 1975.
- [22] Chase, and Simon, "Chess"
- [23] Bonar, J., *Understanding the bugs of novice programming*, Ph.D. Dissertation, Department of Computer and Information Science, University of Massachusetts, Amherst, MA, 1984.
- [24] Johnson, L., and Soloway, E., "PROUST: Knowledge-based program debugging," *Proceedings Eighth International Software Engineering Conference*, Orlando, FL, March 1984.
- [25] Larkin, J., McDermot, J., Simon, D., and Simon, H., "Expert and novice performance in solving physics problems," *Science*, Vol. 208, 20, 1980.
- [26] Reif, F., "Physics....."
- [27] Bonar, J., "Natural problem solving strategies and programming language constructs," *Proceedings of the Fourth Annual Conference of the Cognitive Science Society*, 1982.
- [28] Grice, H., "Logic and Conversants," in Cole & Morgan (eds.), *Syntax and Semantics*, Academic Press, New York, pp. 41-58, 1975.
- [29] collins, A., Warnock, E., and Passafiume, J., "Analysis and synthesis of tutorial dialogues," *Psychology of Learning and Motivation*, Vol. 9, Academic Press, New York, 1975.
- [30] Reichman, R., "Plain Speaking: A Theory and Grammar of Spontaneous Discourse," Ph.D. Thesis, Harvard University, Department of Mathematics, also Bolt, Beranek and Newman, Technical Report #4681, 1981.
- [31] Sass, L., "Parental Communication Deviance and Schizophrenia: A Cognitive-Developmental Analysis," in Vaina & Hintikka (eds.), *Cognitive Constraints on Communication*, 1984.

APPENDIX 5-G

TEACHING A COMPLEX INDUSTRIAL PROCESS

**Beverly Woolf
Darrell Blegen
Johan H. Jansen
Arie Verloop**

COINS Technical Report 86-24

A slightly condensed version of this paper has appeared in the National Conference for Artificial Intelligence (AAAI-86), Philadelphia, PA., 1986.

TEACHING A COMPLEX INDUSTRIAL PROCESS

**Beverly Woolf
Computer and Information Science
University of Massachusetts
Amherst, Massachusetts, 01003**

**Darrell Elegen
Johan H. Jansen
Arie Verloop
J. H. Jansen Co., Inc.
Steam and Power Engineers
18016 140 Ave. N.E.
Woodinville (Seattle), WA 98072**

ABSTRACT

Computer training for industry is often not capable of providing advice custom-tailored for a specific student and a specific learning situation. In this paper we describe an intelligent computer-aided system that provides multiple explanations and tutoring facilities tempered to the individual student in an industrial setting. The tutor is based on a mathematically accurate formulation of the kraft recovery boiler and provides an interactive simulation complete with help, hints, explanations, and tutoring. The approach is extensible to a wide variety of engineering and industrial problems in which the goal is to train an operator to control a complex system and to solve difficult "real time" emergencies.

This work was supported by The American Paper Institute, Inc. a non-profit trade institution for the pulp, paper, and paperboard industry in the United States, Energy and Materials Department, 260 Madison Ave., New York, NY, 10016. Preparation of this paper was supported by the Air Force Systems Command, Rome Air Development Center, Griffiss AFB, New York, 13441 and the Air Force Office of Scientific Research, Bolling AFB, DC 20332 under contract No. F30602-85-C-0008

1. Tutoring Complex Processes

Learning how to control a complex industrial process takes years of practice and training; an operator must comprehend the physical and mathematical formulation of the process and must be skilled in handling a number of unforeseen operating problems and emergencies. Even experienced operators need continuous training. A potentially significant way to train both experienced and student operators for such work is through a "reactive computer environment" [Brown et al., 1982] that simulates the process and allows the learner to propose hypothetical solutions that can be evaluated in "real time". However, a simulation without a tutoring component will not test whether a student has actually improved in his ability to handle the situation. In addition, a simulation alone might not provide the conceptual fidelity [Hollan, 1984] necessary for an operator to learn how to use the concepts and trends of the process or how to reason about the simulation. For instance, evaluating the rate of change of process variables and comparing their relative values over time is an important pedagogical skill supporting expert reasoning; yet rate of change is a difficult concept to represent solely with the gauges in a traditional simulation.

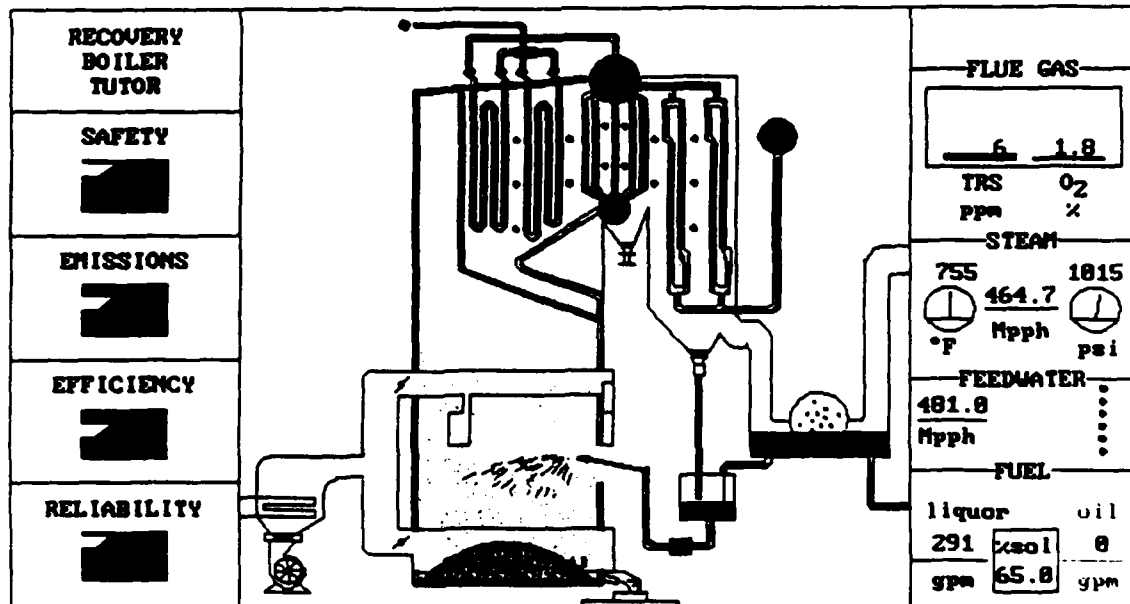


Figure 1: Sectional View of the Recovery Boiler.

We have built a Recovery Boiler Tutor, RBT, that provides tools for developing abstract models of a complex process. The system does not actually represent the mental models that a learner might develop; rather, it provides tools for reasoning about that complex process. These tools include graphs to demonstrate the relationship of process parameters over time, meters to measure safety, emissions, efficiency, reliability, and safety, and interactive dialogues to tutor the operator about the on-going process. The system renders a mathematically and physically accurate simulation of a kraft boiler and interacts with the student about those concepts needed for his exploration of the boiler. Our goal has been to couple the motivational appeal of an interactive simulation with the tutoring and modeling ability of an artificial intelligence system to direct the student in his experimentation.

The tutor was built in direct response to a serious industrial situation. Many industrial accidents, caused in part by human errors, have lead to dangerous and costly explosions of recovery boilers in pulp and paper mills. The American Paper Institute¹ built the interactive tutor to provide on-site training in the control room of recovery boilers. The tutor is now being beta tested in pulp and paper mills across the United States and is being prepared for nationwide distribution.

2. The Recovery Boiler Tutor

A recovery boiler is an extremely complex machine found in hundreds of pulp and paper mills around the world. Figure 1, from the RBT screen, pictures a typical boiler and some of the meters used by an operator to control the process. The actual boiler can cost up to \$70 million and can be up to 20 stories high. Its function is to processes the spent black liquor that can be produced in a pulp mill, up to 1500 tons/day, and to allow recovery and re-use of the pulping chemicals. It is a process in which hot and molten inorganic chemicals may lead to smelt/water and combustible gas explosions, endangering life and property.

The goal behind the tutor is to challenge an operator, to solve boiler problems and to maintain his incentive to continue to "play" with the tutor. The operator is encouraged to move setpoint controllers and to watch the results of his actions on the animated boiler or control board. The tutor monitors the operator's freewheeling trial and error behavior and provides help, hints, explanations, and tutoring when needed or requested.

¹The American Paper Institute is a non-profit trade institution for the pulp, paper, and paperboard industry in the United States.

[Note: A partial blackout is in progress and it will lead to deterioration of the combustion process if no action is taken. Meters indicate that CO, SO₂, and O₂ readings are abnormally high.]

[An alarm buzzer sounds.]

Tutor: What action are you going to take?

Student: Check smelt spout cooling water. [Student's action does not result in a change in the boiler.]

Tutor: It looks as though the liquor isn't burning . . .

Student: Help. What is going on? [Student requests help.]

Tutor: A partial blackout is occurring and this causes TRS, CO, SO₂, and O₂ to increase. Steam flow and steam temperature are dropping. I think part of the bed is not burning.

Student: Rod the primary air ports. [A small improvement in the boiler readings can be observed.]

Tutor: Your action is correct; however, at this time rodding alone is not effective

. . .
Student: Raise the liquor temperature. [Student's action does not reverse the emergency and a new alarm buzzer sounds]

Tutor: Rodding the air port apparently was not enough to undo the blackout.

Student: Increase the oil fuel supply. [Student's action reverses the emergency.]

Tutor: That solved the problem alright. Good thinking.

Analysis of the problem: You had a partial blackout caused by plugged primary air ports and a cold bed. Partial blackout can be effectively treated through a combination of rodding the primary air ports and adding more heat. The problem can be avoided by keeping the air ports clean.

Figure 2: Dialogue Between Tutor and Operator.

An example interaction^o between the student and tutor is shown in Figure 2. As the operator changes setpoint controllers and requests information about the boiler, the tutor selectively discusses the optimality of his actions (we show how in Section 3) and suggests how he might better focus his action or better utilize his data. An important feature to note about this dialogue is that at any point during the simulated emergency there are a large number of actions an operator might take and, as the problem worsens, an increasing number of actions that he *should* take to correct the operating conditions. Thus, an immediate and correct response might require only one action, such as rodding the primary air ports, but

^oThe dialogue of Figure 2 was not actually produced in natural language; student input was handled through menus (Figure 3) and tutor output produced by cutting text from emergency-specific text files loaded when the emergency was invoked.

```

What Are You Going to Do

Determine source of dilution
Check instrumentation
Check dissolving tank agitators
Rod smelt spout
Use portable auxiliary burner
Remove liquor guns
Put in liquor guns
Clean liquor guns
Rod primary air ports
Rod secondary air ports
Check smelt spout cooling water
Start standby feedwater pumps
Restore water flow to deaerator
Quit

```

(A)

```

What Do You Want To Do

Look at boiler
Manually adjust controls
Flip emergency switch
See panelboard
See alarm status
Go do something
See trends
Examine report
Help
Go to analysis & quit
Change RBT's mode
Nothing

```

(B)

Figure 3: Menus to Select Tasks to be Performed on the Boiler.

a delayed response causes the situation to worsen and requires the addition of auxiliary fuel.

The operator interacts with the tutor through a hierarchy of menus, one of which is shown in Figure 3. The first menu, (A), allows an operator to select a physical activity to be performed on the boiler, such as checking for a tube leak or rodding the smelt spout. The second menu, (B), allows the operator to select a particular computer screen, such as the alarm board or control panel board.

The student can initiate any of 20 training situations, emergencies, or operating conditions (see Appendix 1). He can also ask that an emergency be chosen for him or he might accidentally trigger an emergency as a result of his actions on the boiler. Once an emergency has been initiated, the student should adjust meters and perform actions on the simulated boiler to solve the emergency.

For example, if the system has simulated a TRS reading of greater than 15 ppm and if the amount of oxygen is less than 2%, then the student is expected to increase the oxygen until it is 2.5%. If he does this, the level of TRS will automatically be reduced to less than 5 ppm and the boiler will return to a normal state. However, if he does not perform this action, a critical situation will develop accompanied possibly by a blackout and, if the situation is allowed to continue, a dangerous explosion.

While the simulation is running, the operator can view the boiler from many directions and can focus in on several components, such as the fire bed in Figure 4. The tutor provides assistance through visual clues, such as a darkened smelt bed; acoustic clues, ringing alarm buzzers, textual help, explanations, and dialogues, such as that illustrated in Figure 2. The operator can request up to 30 process parameters on the complete panel board, Figure 5 or can view an alarm board (not shown). The tutor allows the student to change 20 setpoints and to ask

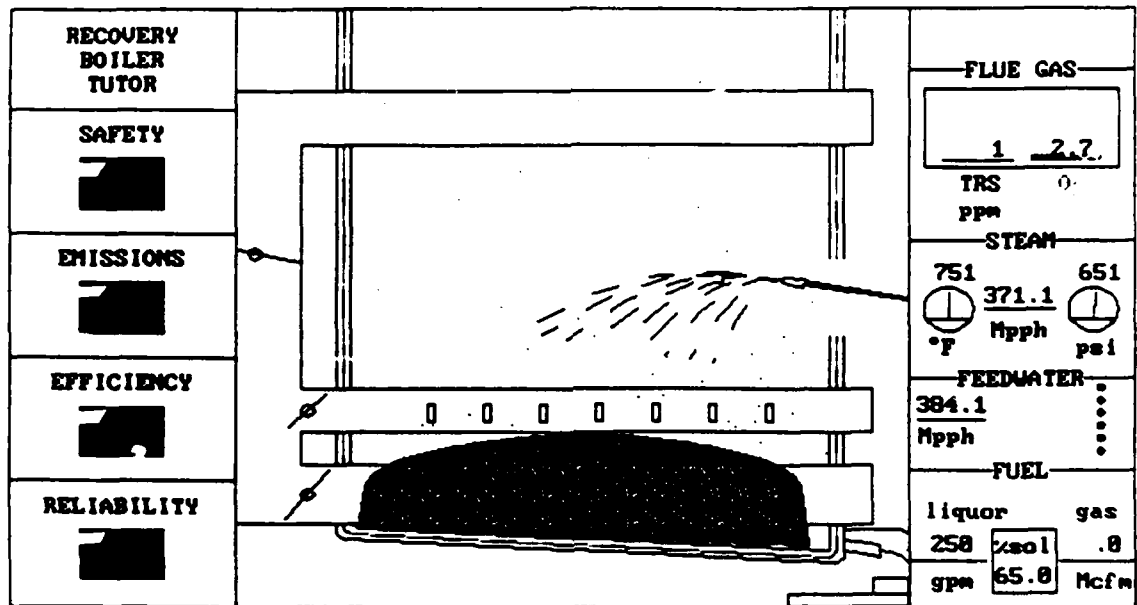


Figure 4: Focused View of the Fire Bed.

menued questions such as "What is the problem?", "How do I get out of it?", "What caused it?", and "What can I do to prevent it?." The operator can request meter readings, physical and chemical reports, dynamic trends of variables. All variables are updated in real time (every 1 or 2 seconds).

In addition to providing information about the explicit variables in the boiler, RBT provides information about implicit processes through *reasoning* tools, with which an operator can understand and reason about the complex processes. One such tool is composite meters (left side of Figures 1 and 5). These meters record the state of the boiler using synthetic measures for safety, emissions, efficiency, and reliability of the boiler. The meter readings are calculated from complex mathematical formulae that would rarely, if ever, be used by an operator to evaluate the same characteristics of their boiler. For instance, the safety meter is a composition of seven independent parameters, including steam pressure, steam flow, steam temperature, feedwater flow, drum water level, firing liquor solids, and combustibles in the flue gas. Meter readings allow a student to make

*These four questions are answered by cutting text from a file which was loaded with the specific emergency. These questions do not provide the basis of the tutor's knowledge representation, which will be discussed in Section 3.2

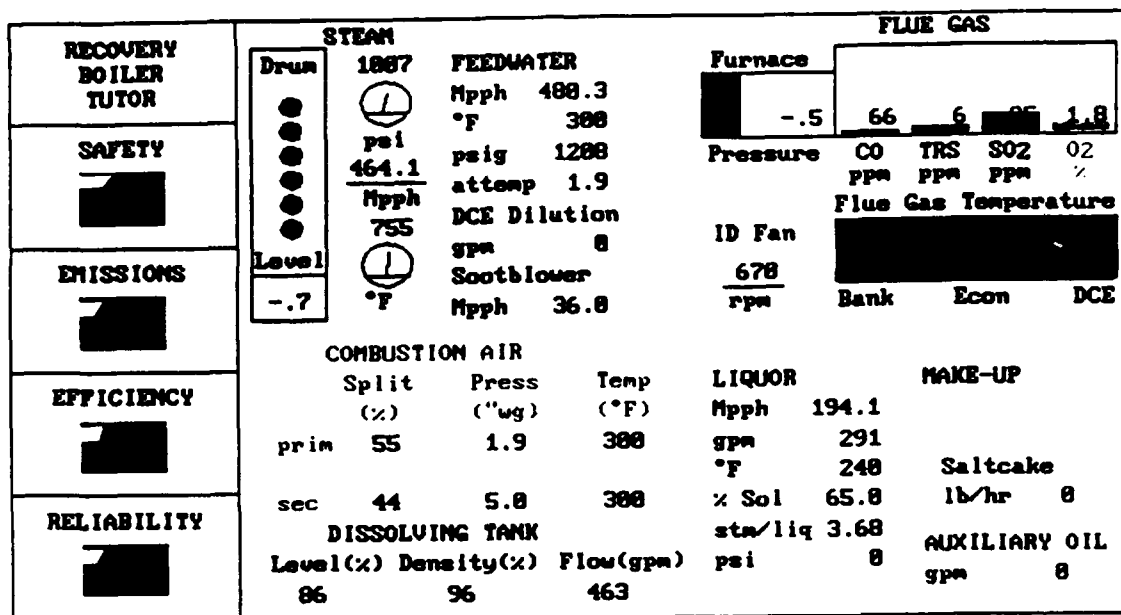


Figure 5: The Complete Control Panel.

inferences about the effect of his actions on the boiler using characteristics of the running boiler. These meters are not presently available on existing pulp and paper mill control panels; however, if they prove effective as training aids, they could be incorporated into actual control panels.

Other reasoning tools include trend analyzers, Figure 6, and animated graphics, such as shown in Figures 1 and 4. Trend analyzers show an operator how essential process variables interact in real time by allowing him to select up to 10 variables, including liquor flow, oil flow, and air flow, etc, and to plot each against the others and time. Animated graphics are provided as a part of every view of the boiler and include realistic and changing drawings of dynamic components of the boiler, such as steam, fire, smoke, black liquor, and fuel.

Each student action, be it a setpoint adjustment or proposed solution, is recorded in an accumulated response value. This value reflects an operator's overall score and how successful, or unsuccessful, his actions have been and whether the actions were performed in sequence with other relevant or irrelevant actions. This accumulated value is not presently used by the tutor, but the notation might be used to sensitize the tutor's future responses to the student's record. For instance, if the operator has successfully solved a number of boiler emergencies, the accumulated value might be used to temper subsequent tutoring

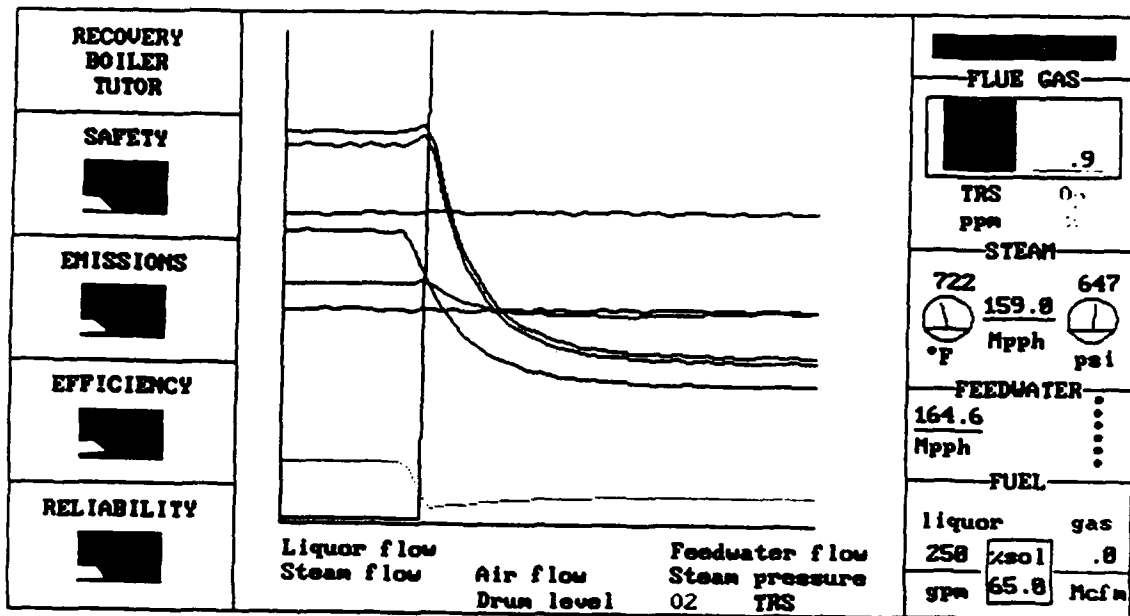


Figure 6: Trends Selected by the Operator.

so that it is less intrusive. Similarly, if a student's past performance has been poor, the accumulated value could be used to activate more aggressive responses from the tutor.

3. Multiple Representations of Knowledge

Multiple concepts and processes were represented in RBT, some procedurally, some declaratively, and some in both ways. For example, emergencies in the steam boiler were first represented as a set of mathematical formulae so that process parameters and meter values could be produced accurately in the simulation. Then these same emergencies were encoded within the tutor's knowledge base as a frame-like data structure with slots for preconditions, optimal actions, and conditions for solution satisfaction so that the tutor could evaluate and comment upon the student's solution.

RBT can recognize and explain:

- equipment and process flows,
- emergencies operating problems as well as normal conditions,
- solutions to emergencies and operating problems,
- processes for implementing solutions, and
- tutoring strategies for assisting the student.

Four modules were used to represent this knowledge: *simulation*, *knowledge base*, *student model*, and *instructional strategies*. Development of the last three components was inspired by prior work in intelligent tutoring systems [Brown et al., 1982; Anderson et al., 1985, Sleeman, 1982, Slater et al., 1985; and Woolf and McDonald, 1984a, 1984b]

3.1 Simulation of Equipment and Process Flow

The *simulation* uses a mathematical foundation to depict processes in a boiler through meter readings and four animated views of the boiler. It reacts to more than 35 process parameters and generates dynamically accurate reports of the thermal, chemical, and environmental performance of the boiler (not shown) upon request. An alarm board (not shown) represents 25 variables whose button will turn red and alarm sounded when an abnormal condition exists for that parameter. The simulation is interactive and inspectable in that it displays a "real time" model of its process, yet allows the student to "stop" the process at anytime to engage in activities needed to develop his mental models [Hollan et al., 1984]. The operators who tested RBT mentioned that they like being able to stop the process to ask questions or to explore boiler characteristics.

If a student working on a problem inadvertently triggers a second problem, the least serious problem will be placed on a stack and held in abeyance while the student is coached to solve the more serious problem. After the more serious problem is solved, the student is coached to solve the remaining one. Thus, the simulation provides facilities for handling multiple instantiations of emergencies.

*Engineering details about the steam and chemical parameters in RBT and the boiler simulation capabilities can be found in [Jansen et al., 1986].

One advantage of a formal representation of the process is the availability of a "database" of possible worlds into which information based on typical or previous moves can be fed into the simulation at anytime [Brown et al., 1982] and a solution found. In this way, a student's hypothetical cases can be proposed, verified, and integrated into his mental model of the boiler.

3.2 Knowledge Base of Emergencies and Operating Conditions

The *knowledge base* contains preconditions, postconditions, and solutions for emergencies or operating conditions, described as scenarios. Scenarios are represented in frame-like text files containing preconditions, postconditions, and acceptable solutions for each scenario. For example, in Lisp notation, a true blackout would be described as:

```
preconditions:
  (or (<= blackout_factor 1)
      (< heat_input 5000))
postconditions:
  (or (increasing O2)
      (decreasing steamflow)
      (increasing TRS)
      (increasing CO)
      (increasing SO2))
solution_satisfaction:
  (and (= blackout_factor 1)
       (> heat_input 5200))
```

Scenarios in RBT have been teased apart to represent successively more serious problems. For instance, a smelt spout pluggage is represented as separate scenarios depending on whether the solution requires rodding the spout, applying a portable auxiliary burner, removing the liquor, or a combination of all three. Again, formalized knowledge of the domain made it easy to represent and evaluate graduated scenarios, as well as multiple operator actions.

The efficiency of the student's action is evaluated both through the type of action performed, such as increasing O₂ or increasing steamflow for a true blackout, and the effect of that action on the boiler. Thus, if an inappropriate action nevertheless resulted in a safe boiler, the student would be told that his action worked, but that it was not optimal. For example, a partial furnace blackout requiring manual rodding of the air delivery system can be alleviated by shutting down the boiler. However, this is an expensive and unwarranted action and the student will be advised to use an alternative approach.

3.3 Student Model to Monitor the Operator's Solution

The *student model* records actions carried out by the student in solving the emergency or operating problem. It recognizes correct as well as incorrect actions and identifies each as relevant, relevant but not optimal, or irrelevant.

The tutor compares the student's actions with those specified by the knowledge base and uses a simplified differential model to recognize and comment about the difference between the two. For instance, if a partial blackout has been simulated, the black liquor solids are less than 58%, and the operator adjusts the primary air pressure, the tutor might interrupt with a message such as:

"Primary air pressure is one factor that might contribute to blackout, but there is another more crucial factor - try again."

or

"You have overlooked a major contributing factor to blackouts."

The student model is currently the weakest component of the tutor. We intend to incorporate inferences about patterns of student errors and possible misconceptions^{*} as a way to increase the tutor's ability to reason about what the operator has accomplished so far and what possible misconception he has. For example, we would like to test presumed misconceptions and use future operator actions to verify the existence of those misconceptions. To do so, the student model would have to link misconceptions with scenarios and to record all common errors and evidence for possible misconceptions.

3.4 Instructional Strategies to Assist the Student

The *instructional strategies* contain decision logic and rules to guide the tutor's intervention of the operator's actions. In RBT, the intent has been to "subordinate teaching to learning" and to allow the student to experiment while developing his own criteria about boiler emergencies. The tutor guides the student, but does not provide a solution as long as the student's performance appears to be moving closer to a precise goal.

*Misconceptions will be compiled by J. H. Jansen Co., Inc. Steam and Power Engineers, who, in addition to being the authors of RBT, have extensive operating experience with boilers in the U.S.A. and Canada.

Represented as if/then rules based on a specific emergency and a specific student action, the instructional rules are designed to verify that the student has "asked" the right questions and has made the correct inferences about the saliency of his data. Responses are divided into three categories:

Redirect student: "Have you considered the rate of increase of O₂?"

"If what you suggest is true, then how would you explain the low emissions reading?"

Synthesize data: "Both O₂ and TRS have abnormal trends."

"Did you notice the relation between steam flow and liquor flow?"

Confirm action: "Yes, It looks like rodding the ports worked this time".

The tutor selects from within each category a response that address both the operator's action and his apparent ability to solve the problem. Special precautionary messages are added to the most specific tutor responses to alert an operator when a full scale disaster is imminent.

The instructional strategies are designed to encourage an operator's generation of hypotheses. Evidence from other problem solving domains, such as medicine [Barrows and Tamblyn, 1980], suggests that students generate multiple (usually 3-5) hypotheses rapidly and make correct diagnoses with only 2/3 of the available data. The RBT tutor was designed to be a partner and co-solver of problems with the operator, who is encouraged to recognize the effect (or lack of same) of his hypotheses and to experiment with multiple explanations of an emergency. No penalty is exacted for slow response or for long periods of trial and error problem solving.

This approach is distinct from that of Anderson et al., [1985] and Reiser et al., [1985] whose geometry and Lisp tutors immediately acknowledge a incorrect student answers and provide hints. These authors argue that erroneous solution paths in geometry and Lisp are often so ambiguous and delayed that they might not be recognized for a long time, if at all, and then the source of the original error might be forgotten. Therefore, immediate computer tutor feedback is needed to avoid fruitless effort.

However, in industrial training, the trainee must learn to evaluate his own performance from its effect on the industrial process. He should trust the process itself to provide the feedback, as much as is possible. In RBT we provide this

*Medical students have been found to ask 60% of their questions while searching for new data and obtain 75% of their significant information within the first 10 minutes after a problem is stated [Barrows and Tamblyn, 1980].

feedback through animated simulations, trend analyses, and "real-time" dynamically updated meters. The textual dialogue from the tutor provides added assurance that the operator has extracted as much information as possible from the data and it establishes a mechanism to redirect him if he has not [Burton and Brown, 1982; Goldstein, 1982].

4. Developmental Issues

RBT was developed on an IBM PC AT (512 KB RAM) with enhanced graphics and a 20 MB hard disk. It uses a math co-processor, two display screens (one color), and a two key mouse. The simulation was implemented in Fortran and took 321 KB; the tutor was implemented in C and took 100 KB.

Although we tried to implement the tutor in Lisp, we found extensive interfacing and memory problems, including segment size restrictions (64k), incompatibility with the existing Fortran simulator, and addressable RAM restrictions (640K). To circumvent these problems the tutor was developed in C with many Lisp features implemented in C, such as functional calls within the parameters of C functions. Meter readings and student actions were transferred from the simulation, in Fortran, to the tutor, in C, through vectors passed between the two programs.

The approach taken here can be extended to other engineering and industrial training problems. Factors that are likely to be considered in building a training system are availability, cost, and appropriateness of software and hardware for the scope of the task. In our case, decisions were made to ensure swift production of a simulation and tutor, given approximately 18 months development time.

5. Evaluations

The tutor has been well-received thus far. It is presently used in actual training in the control rooms of several pulp and paper mills throughout the U.S. Formal evaluation will be available soon. However, informal evaluation suggests that working operators enjoy the simulation and handle it with extreme care. They behave as they might at the actual control panel of the pulp mill, slowly changing parameters, adjusting meters through small intervals, and checking each action and examining several meter readings before moving on to the next action.

Both experienced and novice operators engage in lively use of the system after about a half hour introduction. When several operators interact with the tutor, they sometimes trade "war stories" advising each other about rarely seen situations. In this way, experienced operators frequently become partners with novice operators as they work together to simulate and solve unusual problems.

6. Conclusions

Several fundamental lessons about building an intelligent tutor were learned from this project. The first and foremost was the need for "in-house" expertise; in our case the programmer, project manager, and director of the project were themselves chemical engineers. More than 30 years of theoretical and practical knowledge about boiler design and teaching was incorporated into the system. Had these experts not previously identified the chemical, physical, and thermodynamic characteristics of the boiler and collected examples of successful teaching activities, development time for this project would have been much longer.

A second critical lesson was the need to clarify and implement the components of a teaching philosophy early in the development process in order to ensure full realization of a tutor in the completed system. For example, in order to manifest a philosophy of subordinating teaching to learning, we had to build up the system's ability to recognize partially correct as well as irrelevant actions, (in the knowledge base), to custom-tailor its responses to each type of answer (in the instructional strategies), and to quietly monitor the operator while judiciously reasoning about when to interrupt him (in the student model). The need to limit authoritarian responses from the system and to restrict it to giving only as much help as absolutely needed, meant that tutoring was not tacked onto the end of an expert system, but rather was developed as a part of components of the expert system. We suggest that silence (inactivity) on the part of a computer system is in itself a recognition of the learner's role in the training process and provides an expression of our confidence in his progress.

A third and most surprising lesson learnt from this project was that a teaching system can be designed for multiple students. The system is now being used with groups of operators who work with each other and with the computer to solve problems; pedagogically wholesome things are beginning to happen among them. For example, novice and experienced operators, who might otherwise not be comparable in training and ability, can share their problem solving knowledge and experience; each teaching and learning in a non-evaluative environment.

Several issues remain unresolved in our work to improve the computer tutor's ability to respond to the student. We need to sort out those skills or processes that a student has learned from those that he is still trying to learn and to sort

out those concepts he has from those he still has problems with; we also need to recognize which techniques have been effective in helping him. Currently, the tutor can not do this and we have suggested how we might extend the student model to incorporate inferences made about the student's knowledge, his errors and potential misconceptions to make progress along these lines.

7. Acknowledgements

The authors thank Jeremy Metz, Bradford Leach, and the A.P.I. Recovery Boiler Committee for their encouragement and support.

8. References

- Anderson, J., Boyle, C., and Yost, G., "The Geometry Tutor," in *Proceedings of the International Joint Conference on Artificial Intelligence*, Los Angeles, 1985.
- Barrows, H. S., and Tamblyn, R. H., *Problem-Based Learning: An Approach to Medical Education*, Springer Publishing Co., New York, 1980.
- Burton, R., and Brown, J., "An Investigation of Computer Coaching for Informal Learning Activities," in Sleeman, D. and Brown, J. S. (Eds.), *Intelligent Tutoring Systems*, Academic Press, Cambridge, Mass, 1982.
- Brown, J., Burton, R., and deKleer, J., "Pedagogical Natural Language, and Knowledge Engineering Techniques in SOPHIE I, II, and III," in Sleeman, D. and Brown, J. S. (Eds.), *Intelligent Tutoring Systems*, Academic Press, Cambridge, Mass, 1982.
- Goldstein, I., "The Genetic Graph: A Representation for the Evolution of Procedural Knowledge," in Sleeman, D. and Brown, J. S. (Eds.), *Intelligent Tutoring Systems*, Academic Press, Cambridge, Mass, 1982.
- Hollan, J., Hutchins, E., and Weitzman, L., "STEAMER: An Interactive Inspectable Simulation-based Training System," in *The AI Magazine*, Summer, 1984.
- Jansen, J., Verloop, A., and Blegen, D., "Recovery Boiler Tutor: An Interactive Simulation and Training Aid," in *Proceedings of the Technical Association of the Pulp and Paper Industry Engineering Conference*, Seattle, 1986 (in print).

Reiser, B., Anderson, J., and Farrell, R., "Dynamic Student Modelling in an Intelligent Tutor for Lisp Programming," in *Proceedings of the International Joint Conference on Artificial Intelligence*, Los Angeles, 1985.

Slater, J., Petrossian, R., and Shyam-Sunder, S., "An Expert Tutor for Rigid Body Mechanics: Athena Cats - MACAVITY," in *Proceedings of the Expert Systems in Government Symposium*, IEEE and MITRE Corp, Oct 1984.

Sleeman, D., "Assessing Aspects of Competence in Basic Algebra," in Sleeman, D. and Brown, J. S. (Eds.), *Intelligent Tutoring Systems*, Academic Press, Cambridge, Mass, 1982.

Wolf, B. and McDonald, D., "Context-dependent Transitions in Tutoring Discourse," in *Proceedings of the National Conference on Artificial Intelligence*, (AAAI), Austin, TX, Aug 1984a.

Wolf, B. and McDonald, D., "Design Issues in Building a Computer Tutor," in *IEEE Computer*, Sept 1984b.

The API. Recovery Boiler Reference Manual, Prepared by J. H. Jansen Co., American Paper Institute, New York, NY., 1982.

9. Appendix 1: Emergencies and Operating Problems Simulated by the Tutor

Emergency Situations:

Smelt/Water Explosion
Combustible Gas Explosion
Tube Rupture (various locations)

Operating Problems:

High Drum Water Level
Low Drum Water Level
Loss of Steam Header Pressure
Nozzle Pluggage
Liquor Supply Loss
Smelt Spout Pluggage
Heavy Smelt Run-off
ID Fan Failure
FD Fan Failure
Carryover and Pluggage
Depleted Weak Wash Flow

Low Liquor Firing Solids
Partial Blackout
Complete Blackout
Instrument Air Failure
Electrical Power Failure

Building a Community Memory for Intelligent Tutoring Systems¹

Beverly Woolf and Pat Cunningham†

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

†The Hartford Graduate Center, Hartford, Conn 06101

Abstract

This article discusses the need for multiple experts to work together to develop knowledge representation systems for intelligent tutors. Three case studies are examined in which the need for a pragmatic approach to the problem of knowledge acquisition has become apparent. Example methodologies for building tools for the knowledge acquisition phase are described including specific tasks and criteria that might be used to transfer expertise from several experts to an intelligent tutoring system.

I. A Community Memory

Building intelligent tutoring systems requires community knowledge, i.e., multiple experts working together to encode individual expertise in an intelligent tutor. This knowledge acquisition phase might span months or years. Thus, we need a framework to simplify *changing knowledge* in the tutors as well as a suite of programming tools for browsing and summarizing knowledge, for tracing and explaining the student model, and for tracking reasoning about teaching strategies. In short, tools and methodologies are needed that can be used specifically for knowledge acquisition activities within an intelligent tutor. In this paper we share our experience of building three intelligent tutors and describe the criteria for, and in some cases, the emerging tools used within this acquisition process.

The concept of a community memory for intelligent tutors reflects the fact that knowledge of tutoring is often distributed, incomplete, and acquired incrementally [Bobrow, Mittal and Stefik, 1986] and thus requires contributions from several experts. This is especially true in tutoring systems because the domain expert, cognitive scientist, and teaching expert are typically not the same person. Given multiple experts who contribute to building the system and the need for a large amount of testing and modification to fine tune the tutor, completion of a tutor can not be the "final" step in development of a single system, but rather must be a forcing function between the

completion of one system and the beginning of another. A completed knowledge base provides grit for our collective grinder, forcing us to further clarify and amplify teaching and learning knowledge and to improve communication between those experts who contribute to it.

Articulating and incorporating communal knowledge into a tutor reveals a great deal about each area of expertise and about the tools used by the experts to perform problem solving in the domain. For example, building the boiler tutor described in Section 2.1 indicated several weaknesses in the tools available to industrial boiler operators. We therefore developed simulation tools, including abstract meters and trends (Figure 1) that might ultimately be integrated into the equipment used by boiler operators. Similarly, in building a geometry tutor [Anderson, Boyle, and Yost, 1985] provided an environment that would be a valuable aid to motivated learners, even without help from any on-line tutor. Anderson introduced visualization and forward and backward reasoning templates that would facilitate geometry problem-solving independent of teaching media.

In the next section, we briefly describe our three intelligent tutors and in Section III indicate some methodologies for how knowledge can be acquired from multiple experts to build additional tutors.

II. CASE STUDIES

A. RBT for Teaching Complex Industrial Processes

The first tutor to be discussed is fully implemented, tested, and now used for training in nearly 60 industrial sites across America. The Recovery Boiler Tutor, RBT², is described elsewhere [Woolf, Blegen, Jansen and Verloop, 1986], and will only be summarized here. It provides multiple explanations and tutoring facilities tempered to the individual user, a control room operator. The tutor is based on a mathematically accurate formulation of the boiler and provides an interactive simulation, (Figure 1) complete with help, hints, explanations, and tutoring.

¹This work was supported in part by the Air Force Systems Command, Rome Air Development Center, Griffiss AFB, New York, 13441 and the Air Force Office of Scientific Research, Bolling AFB, DC 20332 under contract No. F30802-85-C-0008. This contract supports the Northeast Artificial Intelligence Consortium (NAIC). Partial support also from URI University Research Initiative Contract No. N00014-86-K-0764.

²RBT was built by J. H. Jansen Co., Inc., Steam and Power Engineers, Woodinville (Seattle) Washington and sponsored by The American Paper Institute, a non-profit trade institution for the pulp, paper, and paperboard industry in the United States, Energy Materials Department, 260 Madison Ave., New York, NY, 10016.

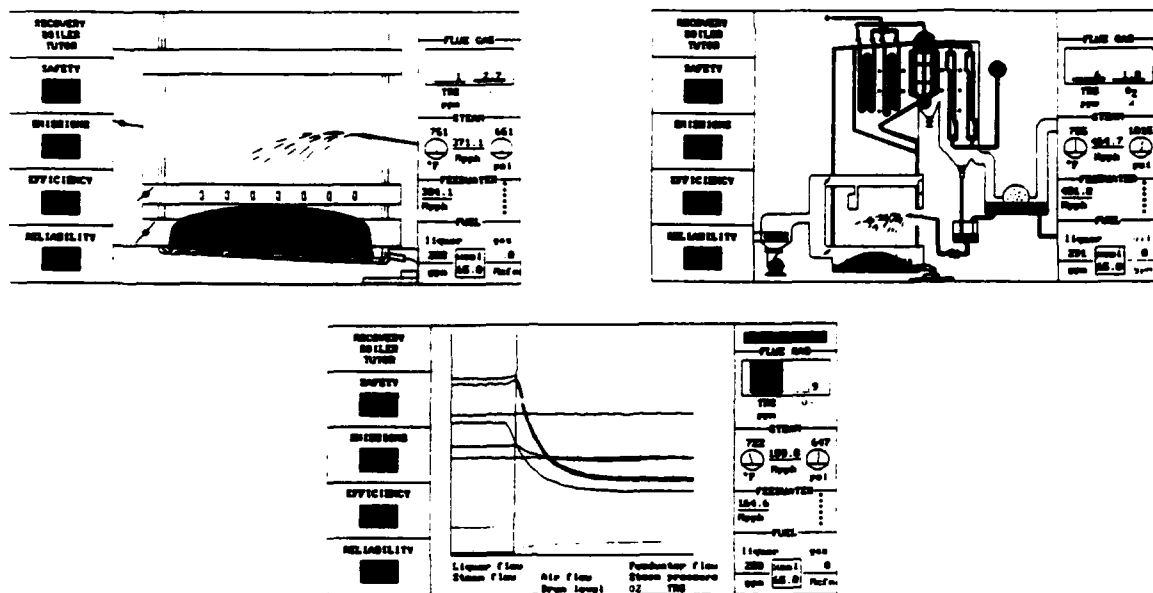


Figure 1: Several Views of the Recovery Boiler Tutor

The tutor challenges operators to solve boiler emergencies while monitoring their actions and advising them about the optimality of their solutions. The tutor recognizes less than optimal and clearly irrelevant actions and modifies its response accordingly. Operators can continue their freewheeling or purposeful problem-solving behavior while the tutor offers help, hints, explanations, and tutoring advice when needed or when requested. Operators gain experience in recognizing the impact of their actions on the simulated boiler and to react before the tutor advises them regarding potential problems.

Meters, as shown on the left side of screens in Figure 1, record the state of the boiler using synthetic measures for *safety*, *emissions*, *efficiency*, and *reliability* of the boiler. The meter readings are calculated from complex mathematical formulas that would rarely (if ever) be used by operators to evaluate the boiler. The meters have already proved effective as training aids in industrial training sites and could possibly be incorporated into actual control panels.

Operators have reported using the system as much as 70 hours in three months to practice solving emergencies. They handle the simulation with extreme care, behaving as they might if they were in actual control of the pulp mill panel, slowly changing parameters, checking each action, and examining several meter readings before moving on to the next action.

B. Caleb for Teaching a Second Language

Our second intelligent tutor teaches languages based on a powerful pedagogy called the "silent way"—a method developed by Caleb Gattegno. The system uses non-verbal

communications within a controlled environment to teach Spanish [Cunningham, 1986]. It uses graphical Cuisenaire rods³, to generate linguistic situations in which the rod plays various roles. For example, it is used as an object to be given or taken by a student, or it is used to brush teeth. As a new rod is presented, the student theorizes about what situation is encountered and types the appropriate phrase below the picture. In the case illustrated at the top of Figure 2 the tutor presents a rod in the center box. The student responds by typing the word for the new piece at the cursor. In the bottom figure, the tutor corrects a student who places an adjective before rather than after a noun. In this exercise, students might have classified the word "blanca" as an adjective referring to the size of the rod before knowing its meaning. The tutor does not clarify students' conjectures. Students can later change a hypothetical definition if in fact the new word turns out to define the color of the rod. Meanwhile, they will have learned to write the word, spell it, and place it correctly in a sentence.

C. ESE for Teaching Physics

A third tutor is now in the early implementation stage. It is part of a program to develop interactive and monitored simulations to teach physics at the high school or college level.⁴ One of these tutors teaches the second law

³Originally developed by Gattegno for teaching arithmetic

⁴These tutors are being built by the Exploring Systems Earth (ESE) consortium, a group of three universities working together to develop intelligent tutors. The schools include the University of Massachusetts, San Francisco State University, and the University of Hawaii.

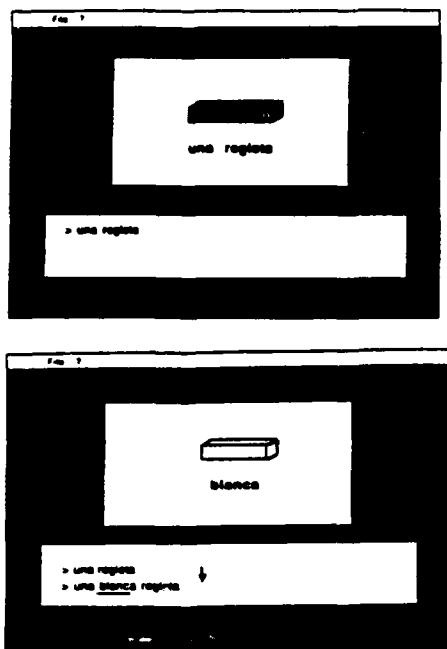


Figure 2: Caleb: A System for Teaching Second Languages

of thermodynamics⁵ and provides a rich environment at the atomic level through which the principles of equilibrium, entropy, and thermal diffusion can be observed and tested [Atkins, 1982]. Students are shown (and are able to construct) collections of atoms that transfer heat to other atoms through random collision (see Figure 3). They can create areas of high-energy atoms, indicated by dark squares, along with variously shaped regions within which the high energy atoms can be monitored. Concepts such as temperature, energy density, and thermal equilibrium can be plotted against each other and against time.

The tutor uses all student activities - including questions, responses, and requests - to formulate its next teaching goal and activity. It uses student actions to determine whether to show an extreme or near-miss example, whether to give an analogy or whether to ask a question. To refine the tutor's response, we are now studying student misconceptions and common errors in learning thermodynamics and statistics.

III. Tools for Knowledge Acquisition

Given the complex heterogeneous nature of the knowledge required to build each of these systems, we need methodologies and tools to transfer teaching and learning knowledge from human experts to systems under construction. Few such tools exist.

⁵The second law states that heat cannot be absorbed from a reservoir and completely converted into mechanical work.

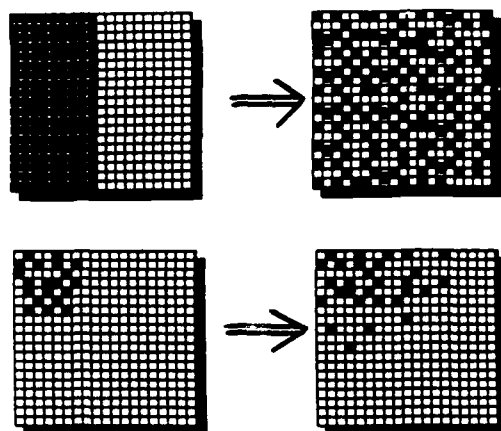


Figure 3: Systems Moving Towards Equilibrium

Expert system shells contain a framework for building knowledge bases about concepts and rules and for making inferences about them. However, they are limited as specific tools for designing and storing tutoring knowledge. They are frequently based on production rules and are limited in representing history and dependency of the tutoring interaction. Also, they inadequately represent tutoring and misconception knowledge such as how to reason about teaching strategies, how to update and assess student models, how to select a path through domain concepts, and how to remediate for misconceptions. In this section, we describe the criteria for developing tools specific to this knowledge acquisition process.

A. Environment Expert

The first expert needed to build an intelligent tutor is the environmental expert. This person often uses a majority of system memory [Bobrow, Mittal and Stefik, 1986] to provide an envelope within which students and system interact. The environment provides specific tools and operators for solving domain problems or for performing domain activities.

Environmental, teaching, cognitive, and domain expert contributions interact strongly with each other - especially those from the environmental expert. For example, a system that asks students to record entrance and exit angles for light in an optics experiment, assumes that the environment supplies such measuring devices.

The following criteria for developing a tutoring environment have begun to emerge:

- 1) Environments should be intuitive, obvious, and fun. Student energy should be spent learning the material, not learning how to use the environment [Cunningham, 1986].

For example, to indicate errors, express feelings or convey meaning, the second-language tutors, visual activities mimic the human Silent Way teachers' gestures, facial expressions, and rods.

2) Environments should record not only what students do, but what they did, intended to do, might have forgotten to do, or were unable to do [Burton, in press]. Environments should provide a "wide bandwidth" within which multiple student activities can be entered and analyzed. For example, the Pascal tutor developed by Johnson and Soloway [1984] processed and analyzed an entire student program before offering advice.

3) Environments should be motivated by teaching and cognitive knowledge about how experts perform tasks and the nature of those tasks. For example, Anderson [1981] performed extensive research with geometry students before developing his geometry tutor interface, and Woolf et al. [1986] incorporated knowledge from experts with more than 30 years experience working with boiler operations before building the RBT interface.

4) Environments must maintain physical fidelity⁶ [Hollan, Hutchins and Weitzman, 1984]. The RBT tutor presents a mathematically exact duplicate of the industrial process. It models and updates over 100 parameters every two seconds. Visual components of the industrial process such as alarm boards, control panels, dials, and reports are duplicated from the actual control room.

5) Environments should be *responsive*, *permissive*, and *consistent* [Apple, 1985]. They should target applications based on skills that people already have, such as moving icons, rather than forcing people to learn new skills. By *responsive*, we mean that student actions should have direct results—that students need not perform rigid sets of actions in rigid and unspecific order to achieve goals. By *permissive*, we mean that students may do anything reasonable and that multiple ways should exist for taking action. By *consistent*, we mean that moving from one application to another, (for example, from editing text to developing graphics), should not require learning new interfaces. All tools should be based on similar interface devices, such as pull-down menus or single and double mouse clicks.

No one environment is appropriate for every domain. We must study each domain to determine how experts function in that domain, how novices might behave differently, and how novices can be helped to attain expert behavior.

B. Teaching Expert

Acquiring sufficient and correct teaching expertise is a long term problem for builders of tutoring systems—in part, because sophisticated knowledge about learning, teaching, and domain knowledge remains an active area of research in most domains. Teaching expertise includes de-

⁶Fidelity measures how closely simulated environments match the real world. High fidelity identifies a system as almost indistinguishable from the real world.

cision logic and rules that guide the tutor's intervention with the student. Tools to facilitate teasing apart and encoding teaching knowledge are just beginning to emerge. For example, we have developed a framework for managing discourse in an intelligent tutor [Woolf and Murray, 1987] that reasons dynamically about discourse, student response, and tutor moves.

The framework (Figure 4) reasons about which pedagogical response to produce and which alternative discourse move to make. It custom-tailors the tutor's response in the form of examples, analogies, and simulations. Discourse *schemas*, or collections of activities and response profiles, are responsible for actually generating system actions and for interpreting student behavior. The number and type of schemas used is dependent on context.

We used empirical criteria to define discourse schemas: tutoring responses were analyzed from empirical studies of teaching and learning and from general rules of discourse structure [Grosz and Sidner].

The framework is flexible and domain-independent; it is designed to be rebuilt—decision points and machine actions are modifiable for fine-tuning system response.

We are now using this framework to improve the physics tutor's response to idiosyncratic student behavior. Response decisions and machine actions, explicitly represented in the system, can be modified through a editor. Appropriate machine response can be assessed continuously and improved. In the long term, we intend to make this reasoning process available to human teachers, who can then modify the tutor for use in a classroom.

No single teaching strategy is appropriate for every domain. For example, Anderson et al. [1985] built geometry and Lisp tutors that responded immediately to incorrect student answers. These authors argued that immediate computer feedback was needed to avoid fruitless student effort.

This pedagogy was opposite to that used by Cunningham [1986] and Woolf et al. [1986]. These latter tutor's advice was passive, not intrusive. The strategy was to subordinate teaching to learning, and to allow students to experiment while developing hypotheses about the domain. The tutors guided their students toward developing their own intuitions, but did not correct them so long as their performance appeared to be attaining a precise goal.

In industrial settings, particularly, trainees must learn to generate multiple hypotheses and to evaluate their own performance based on how their actions affect the industrial process. For example, no human tutor is available during normal boiler operation.

C. Cognitive Expert

At present, the role of the cognitive scientist is incompletely understood; in part, this expert seeks to discover how people learn and teach in a given domain. For example, cognitive science research in thermodynamics will enable systems to recognize common errors, tease apart probable misconceptions, and provide effective remedia-

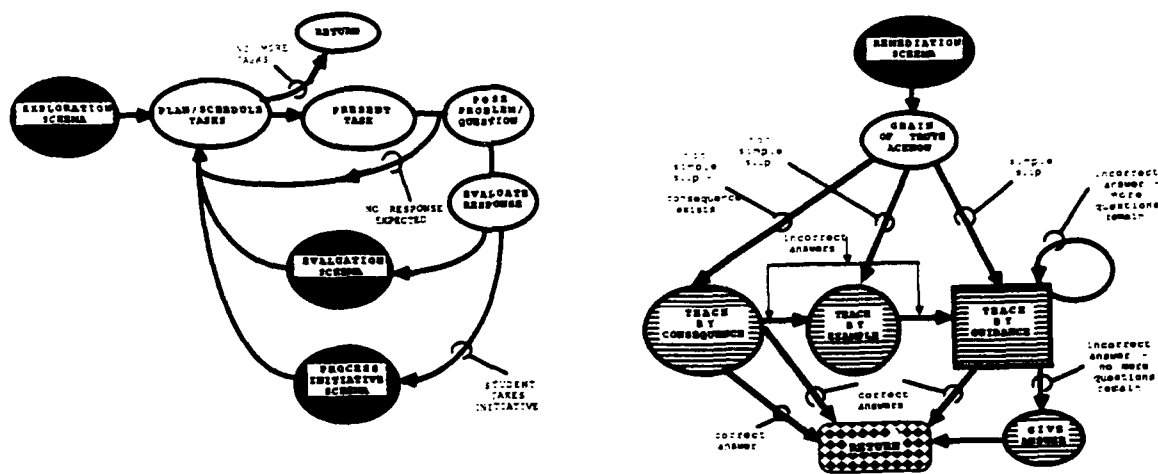


Figure 4: A Framework for Managing Tutoring Discourse

tion. Cognitive science research provides the tutor with a basis for selecting instructional strategies. The importance of addressing common errors and misconceptions in physics is well documented, and the tutor's intelligence hinges on making that knowledge explicit.

We want a tutoring system to help students generate those hypotheses that are necessary precursors to expanding their intuition, and developing their own models of the physical world discover and "listen to" their own scientific intuitions. To do this, we rely on work done by cognitive scientists who study how students reason about qualitative processes, how teachers impart propaedeutic principles (or the knowledge needed for learning some art or science) [Half, in press], and what tools are being used by experts working in the field.

For example, the cognitive science experiments that must be performed to build our thermodynamics tutor include (1) investigation of real-world tools currently used by physicists, (2) examination of studies that focus on cognitive processes used by novices and experts, and (3) comparison of novice with expert understanding of thermodynamics.

RBT articulates cognitive knowledge by explicitly recording student attempts to solve emergencies. It shows students their false paths and gives reasons behind particular rule-of-thumb knowledge used to solve problems. RBT also provides students with various examples from which they can explore problem-solving activities—perhaps in time showing students their own underlying cognitive processes. By using such knowledge, a tutor can begin to help students *learn how to learn*.

D. Domain Expert

An in-house domain expert is critical to building an intelligent tutoring system. By "in-house", we mean that the domain expert must join the project team for anywhere from six months to several years while domain knowledge is being acquired. Any less commitment than that of full-

fledged team member suggests a less than adequate transfer of domain knowledge.

In the tutors described above, the domain experts were (and are) integral to the programming effort. The programmer, project manager, and director of RBT were themselves chemical engineers. More than 30 years of theoretical and practical knowledge about boiler design and teaching strategies were incorporated into the system. Development time for this project would have been much longer than 18 months if these experts had not previously identified the boiler's chemical, physical, and thermodynamic characteristics and collected examples of successful teaching activities.

The second language tutor was developed by a person who holds a graduate degree in teaching English as a second language and has spent more than 7 years using the Silent Way to teach intensive English courses to foreigners living in America and to teach Nepali to American Peace Corps volunteers living in Nepal.

Based on the numerous expert systems projects, the following criteria for acquiring domain knowledge are well understood:

1) Domain experts should be true experts—if possible, the best in the field [Bobrow, Mittal and Stefik, 1986].

2) Domain experts are expensive. Gaining the attention of knowledgeable people is expensive and time consuming. However, the willingness and availability of such experts to participate is critical to the knowledge-engineering process. Assigning the task to a person of lesser ability (or worse, to persons with "time on their hands") might doom a project to failure.

3) Individual domain experts may have incomplete knowledge or conceptual vacuums; therefore multiple experts are needed for testing and modifying domain knowledge throughout the tutor's life.

4) Similarly, domain knowledge can be overly distributed and spread so diffusely among different experts as

to leave severely restricted any system that uses only a single expert [Bobrow, Mittal and Stefik, 1986]. Thus domain knowledge must be acquired incrementally and must be prototyped, refined, augmented and reimplemented. The time needed to build a tutoring system "should be measured in years, not months, and in tens of worker-years, not worker-months" [Bobrow, Mittal and Stefik, 1986].

5) Domain knowledge as found in textbooks is incomplete and idealized [Bobrow, Mittal and Stefik, 1986]. Textbooks rarely contain the commonsense knowledge—the know-how used by expert tutors or professionals in the field—to help choose another teaching strategy or solve difficult problems. Books tend to present clean, uncomplicated concepts and results. To teach or solve real-world problems, tutors must know messy but necessary details of real or perceived links between concepts and unpublished rules of teaching and learning.

IV. Conclusion

Communities of experts are needed to provide a focus for articulating distributed knowledge in an intelligent tutor. The resultant machine tutor should include recent as well as historical research about thinking, teaching, and learning in the domain. Evaluating such an articulation would, in itself, contribute to education—and ultimately to communication between experts.

Compiling diverse research results from environmental, teaching, cognitive, and domain experts is currently hampered by lack of explicit tools to help authors transfer their knowledge to a system. Based on criteria set out above, we intend to continue to develop and integrate knowledge acquisition tools to facilitate assimilation of teaching and learning knowledge into intelligent tutors.

References

- [Anderson, 1981] Anderson, J., Tuning of Search of the Problem Space for Geometry Proofs, *International Joint Conference on Artificial Intelligence*, British Columbia, 1981.
- [Anderson, Boyle, and Yost, 1985] Anderson, J., Boyle, C., and Yost, G., The Geometry Tutor, in *Proceedings of the International Joint Conference on Artificial Intelligence*, Los Angeles, 1985.
- [Apple, 1985] Apple Corp. *Inside Macintosh*, Vol. 1, Addison-Wesley, Reading, Mass., 1985.
- [Atkins, 1982] Atkins, T. *The Second Law*, Freedman Publishers, San Francisco, CA, Scientific American Series, 1982.
- [Bobrow, Mittal and Stefik, 1986] Bobrow, D., Mittal, S., and Stefik, M., Expert Systems: Perils and Promise, in *Communications of the ACM*, Vol 29, #9, 1986.
- [Burton, in press] Burton, R., Instructional Environments, in Polson, M. and Richardson, J. (Eds.) *Foundations of Instructional Tutoring Systems*, Lawrence Earlbaum Associates, Hillsdale, NJ, in press.
- [Cunningham, 1986] Cunningham, P., *Caleb: A Silent Second Language Tutor: The Knowledge Acquisition Phase*, Master's thesis, Tech. Report #87-6, Rensselaer Polytechnic Institute, Troy, NY, 1986.
- [Grosz and Sidner] Grosz, B., and Sidner, C., The Structures of Discourse Structure, *Proceedings of the American Association of Artificial Intelligence*, 1985.
- [Johnson and Soloway, 1984] Johnson L. and Soloway E., Intention-Based Diagnosis of Programming Errors, in *Proceedings of the American Association of Artificial Intelligence, AAAI-84*, Austin, TX, 1984.
- [Half, in press] Half, H., Curriculum and Instruction in Automated Tutors, in Polson, M. and Richardson, J. (Eds.) *Foundations of Instructional Tutoring Systems*, Lawrence Earlbaum Associates, Hillsdale, NJ, in press.
- [Hollan, Hutchins and Weitzman, 1984] Hollan, J., Hutchins, E., and Weitzman, L., STEAMER: An Interactive Inspectable Simulation-based Training System, in *The A.I. Magazine*, 1984.
- [Mittal and Dym, 1985] Mittal, S. & Dym, C., Knowledge Acquisition from Multiple Experts, in *AI Magazine*, 6(2), 1985.
- [Woolf and Murray, 1987] Woolf, B., and Murray, T., A Framework for Representing Tutorial Discourse, *International Joint Conference on Artificial Intelligence*, Milan, Italy, 1987.
- [Woolf, Blegen, Jansen and Verloop, 1986] Woolf, B., Blegen, D., Jansen, J., and Verloop, A., Teaching a Complex Industrial Process, *Proceedings of National Association of Artificial Intelligence*, Philadelphia, PA, 1986.
- [Woolf and McDonald, 1984] Woolf, B., and McDonald, D., Design Issues in Building a Computer Tutor, in *IEEE Computer*, 1985.

A REPRESENTATION FOR COLLECTIONS OF TEMPORAL INTERVALS*

Bruce Leban, David D. McDonald and David R. Forster

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

ABSTRACT

Temporal representation and reasoning are necessary components of systems that consider events that occur in the real world. This work explores ways of considering collections of intervals of time. This line of research is motivated by related work being done by our research group on appointment scheduling and time management. Natural language expressions that refer to collections of intervals are used naturally and routinely in these contexts, and an effective means of representing them is essential.

Previous studies, which considered intervals primarily in isolation, have difficulties in representing some classes of expressions. This occurs not only with expressions that explicitly refer to collections of intervals, such as "the first of every month," but also with expressions that do so only implicitly, such as the U.S. Election Day: "the first Tuesday after the first Monday in November." The traditional solution to this problem has been to provide special means of specifying those forms that are judged to be the most useful (to the exclusion of all other forms).

The "collection representation" builds on previous work in temporal representation by introducing operators that allow the representation of collections of intervals, whether they occur explicitly or implicitly in the expression.

The operators introduced are natural extensions of the relations and operations on intervals. The representation has potential use in scheduling in three areas: graphical display, natural language translation, and reasoning.

I PRIOR WORK

Much of the work on time has focused on temporal reasoning (as opposed to temporal representation). For example, Reacher and Urquhart (1971) and van Benthem (1983) describe temporal logics for reasoning mathemati-

* This work was supported in part by the Air Force Systems Control, Rome Air Force Development Center, Griffiss AFB, New York, 13441 and the Air Force Office of Scientific Research, Bolling AFB, DC 20332 under Contract No. F30602-85-C-0008 and by the National Science Foundation under Support and Maintenance Grant DCR-8318776.

cally about time. The logics are based on the concept that instead of a predicate calculus statement being universally true or false, it may be true or false at different moments of time. Temporal quantifiers (much like the universal and existential quantifiers) are used to augment the calculus.

Allen (1983) describes a computational approach to maintaining knowledge about events in time, for use in AI systems that reason about temporal knowledge. Allen's representation takes the concept of a temporal interval as a primitive and explicitly allows representations of indefinite and relative temporal knowledge. A temporal interval is used as the primitive unit because reasoning about points in time frequently yields counter-intuitive or paradoxical results.

Ladkin (1985, 1986a) makes an argument for the use of non-convex intervals for reasoning. A convex interval is an interval in the usual sense: a contiguous period of time. A non-convex interval is an arbitrary union of convex intervals.

In this paper, it is assumed that a temporal structure based on convex intervals has been defined that has a useful set of operations and relations (see appendix). We believe that the work could be extended to temporal structures based on time-points or non-convex intervals.

II COLLECTIONS OF INTERVALS

An interval t is denoted by $\langle t_\alpha, t_\beta \rangle$ or $\langle t_\alpha; t_\beta \rangle$ where t_α , t_β and $t_\alpha + t_\beta$ are real numbers denoting moments in time; the interval starts at time t_α and extends through time t_β or $t_\alpha + t_\beta$.**

A collection of intervals is a structured set of intervals. The order of a collection is a measure of the depth of the structure. An order 1 collection is an ordered list of intervals. This is somewhat similar to a non-convex interval except that the maximal convex subintervals of

** We ignore the sticky questions of whether the intervals are open or closed and whether time is represented in a continuous or discrete fashion, as these issues are largely irrelevant to the work discussed here. We assume that if t and u are intervals and $t_\beta = u_\alpha$, then $t \cup u = \langle t_\alpha, u_\beta \rangle$.

a non-convex interval are disjoint and the order they are given in is immaterial. An order n collection ($n > 1$) is an ordered list of order $n-1$ collections. The notation used for collections is essentially set notation, except for the understanding that the order of elements is maintained. For example,

$$\{\{x_1, x_2\}, \{x_3, x_4\}, \{x_5\}\}$$

is an order 2 collection. The collection of Thursdays (which contains all the Thursdays in order) is an example of an order 1 collection. The collection of months where each month is represented by a collection of the days in that month (in order) is an order 2 collection.

A. A Formula Approach

Many useful collections can be described by arithmetical formulae, but there are subtle difficulties with this. We reject this approach for the reasons outlined in this section.

Given an appropriate definition for *day* representing the length of one day and, for convenience, assume that time t_0 is Saturday, December 31, 1904, midnight, the collection of Thursdays can be described by the formula:

$$\text{Thursdays} = \{(\alpha; 1 \text{ day}) \mid \alpha = 5 \text{ days} + t_0 \pmod{7 \text{ days}}\}$$

We can generalize *Thursdays* by replacing the 5 with any other value. In other words, it can be understood that *Tuesdays* is an essentially similar collection to *Thursdays*.

The same approach applied to construct the collection of all *Januaries* is less successful. Since every fourth year is a different length, one possible formula is:

$$\text{Januaries} = \{(\alpha; 31 \text{ days}) \mid (\alpha + t_0 \pmod{1461 \text{ days}}) \in \{0, 365, 730, 1095\}\}$$

This formula is considerably more complicated than the one given for *Thursdays*.^{*} More importantly, it fails to provide a means of conveniently recognizing *Augusts* as a generalization of *Januaries*. To generalize from *Januaries* we would need to replace each of the values (except 1461) with appropriate new values: the chance of an arbitrary substitution producing a reasonable generalization is quite small. Essentially, the formula is in a "compiled" form that is quite distant from how the concept would naturally be expressed.

The formulae become even more complicated when new collections must be built from existing collections. For example, consider "the first Thursday of every January." This requires combining the collection of Thursdays and the collection of Januaries to produce a new collection. Furthermore, the system must allow for collections to be combined in fairly arbitrary ways, since it will not be possible to predict all useful specifications.

^{*} It would be even more complicated if it were correct: the Gregorian calendar specifies that only 97 out of every 400 years are leap years

III THE COLLECTION REPRESENTATION

The foundation of the collection representation is a set of primitive collections called *calendars*. A calendar is a collection consisting of an infinite sequence of intervals that span the timeline, i.e., t_i meets t_{i+1} for any two consecutive intervals. A calendar may have a first interval (the first moment in time the system is prepared to consider), but does not have a last interval. *Days, Months and Chinese-Calendar-Years* are instances of calendars.

Two new classes of operators, *slicing* and *dicing*, are defined to operate on collections of intervals. The dicing operators provide means of generating collections from intervals, for example, to break a collection of intervals into smaller intervals. In Figure 1, a dicing operation is illustrated between the first two steps. This operation replaces each interval on the left (a week) with a collection of subintervals (the days in that week).

The slicing operators provide means of selecting intervals from collections of intervals, for example, to select the first interval of a collection. In Figure 1, a slicing operation is illustrated between the second two steps. This operation replaces each order 1 collection (a collection of the days in each week) with a single interval (the fifth day of each week).

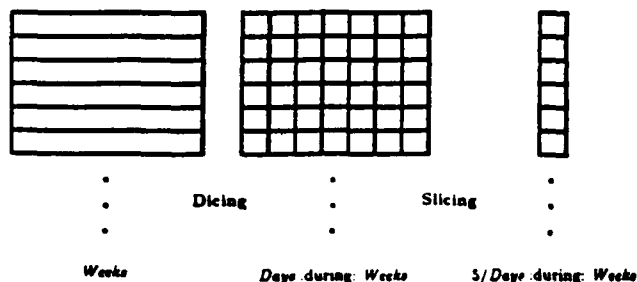


Figure 1. Slicing and Dicing

The terms "slicing" and "dicing" are chosen for both their euphonic and metaphoric appeal.^{**} The operators have a right-to-left precedence. Each operator corresponds roughly to a preposition, so these expressions can be read naturally by someone who speaks a prepositional language (e.g., English).

New collections can be built by combining other collections using these operators. The calendars serve as a basis for this construction. Since the calendars are not sufficient for reasoning about statements that reference collections that might not yet have been defined or might include unknown intervals in the future (e.g., "when Diana is at work"), collections can also be built by predicate reference.

^{**} If these terms seem to have conflicting meanings, "Slicing" can be thought of as corresponding to "Selection" and "Dicing" to "Dividing up"

A. Primitive Collections

A calendar is defined by specifying the intervals of which it is composed. The notation $\langle\langle \alpha; \delta_1; \delta_2; \dots; \delta_n \rangle\rangle$ denotes the calendar

$$\{(\alpha; \delta_1), (\alpha + \delta_1; \delta_2), \dots, (\alpha + \sum_{i \leq n-1} \delta_i, \delta_n), (\alpha + \sum_{i \leq n} \delta_i, \delta_1), \dots\}$$

The list of δ -values is treated as if it were a circular list.

A calendar can also be defined by specifying how it is to be constructed from another calendar. This is denoted by $\langle\langle C; s_1; s_2; \dots; s_n \rangle\rangle$ to indicate that the first interval of this calendar is the union of the first s_1 intervals of C ; the second interval is the union of the next s_2 intervals of C , etc. As above, the list of s -values is treated as a circular list.

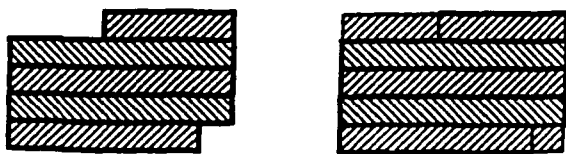
If we assume that the unit of measure is 1 second, we might have the following definitions:

$$\text{Days} \equiv \langle\langle t_0; 86400 \rangle\rangle$$

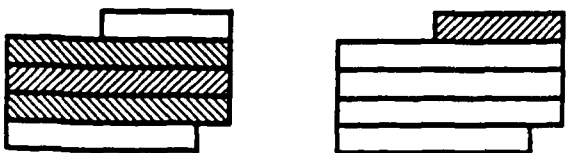
$$\begin{aligned} \text{Months} \equiv \langle\langle \text{Days}; 31; 28; 31; 30; 31; 30; 31; 31; 30; 31; 30; 31; \\ 31; 28; 31; 30; 31; 30; 31; 31; 30; 31; 30; 31; \\ 31; 29; 31; 30; 31; 30; 31; 31; 30; 31; 30; 31 \rangle\rangle \end{aligned}$$

These definitions are intensional rather than extensional. That is, while a calendar defines an infinite data structure, it does not require that an implementation actually build the complete structure, but only that it build those portions of the structure it needs.

Collections can also be constructed from a predicate. The collection $\langle\langle \text{Condition} \rangle\rangle$ is the minimal collection of intervals C that satisfies the property that there does not exist an interval t disjoint from C , such that Condition is true during t . This definition is carefully constructed to avoid the question of whether the predicate operates on intervals or points.



a. *Weeks:overlaps*: (January-1986) b. *Weeks:overlaps*: (January-1986)



c. *Weeks:during*: (January-1986) d. *Weeks:≤*: (January-1986)

Figure 2. Dicing Operators

B. The Dicing Operations

The dicing operators are extensions of the relations on intervals (listed in the appendix). A dicing operator takes an order 1 collection as its left argument, an interval as its right argument and produces an order 1 collection as a result. A dicing operator can also take a collection as the right argument, in which case it operates on each interval in that collection.

For each relational operator (R) there are two dicing operators: strict ($:R:$) and relaxed ($.R.$). If C is an order 1 collection and t is an interval, the dicing operators are defined by:

$$C : R : t = \{c \mid t \mid c \mid c \in C \wedge c R t\} \setminus \{e\}$$

$$C . R . t = \{c \mid c \in C \wedge c R t\} \setminus \{e\}$$

The effect of a strict dicing operator is to break up t into pieces according to C . An illustrative example occurs when C is a calendar. The expression *Weeks:overlaps*: (January-1986) will break up the month on the boundaries of the weeks, i.e., it will give those weeks or parts of weeks that overlap the month. (See Figure 2a.) The effect of a relaxed dicing operator is to select intervals from C that have the appropriate relation with t . Thus *Weeks:overlaps*: (January-1986) will break up the month in the same way as above, but for the weeks at the beginning and end of the month it will give the entire week (including that part not overlapping the month). (See Figure 2b.) In contrast, *Weeks:during*: (January-1986) will give only the weeks that are completely contained in the month. (See Figure 2c.) Finally, *Weeks:≤*: (January-1986) will give only the partial week at the beginning of the month. (See Figure 2d.)

C. The Slicing Operations

The slicing operators, denoted f/C and $|f|/C$, operate on any collection, replacing each of the contained order 1 collections with the result of the application of the slicing operator. Operating on an order 1 collection yields either a single interval or an order 1 collection (usually a subcollection of the original order 1 collection). The expression f/C applies the selection function f to the collection and returns a single interval, while $|f|/C$ returns a collection. f may be a predicate, in which case it constructs a collection containing the intervals which satisfy the predicate. The expression $|f_1, f_2, \dots, f_n|/C$ is the collection consisting of the individual applications of f_1, f_2, \dots, f_n to C in order.

In some cases, a selection function may not have a result (e.g., the 29ths of Februarys), in which case the result is defined to be the empty interval e . Note that since the dicing operators will never produce a collection that contains e , any result that includes e is a sign of a failed selection operation.

The integers are defined as selection functions so that n/C selects the n th interval in C and $-n/C$ selects the n th interval from the end. The function the is defined so that the/C selects the single interval of C , and produces e if C contains other than a single interval.

The function any is used to select intervals nondeterministically. any/C selects a single interval of C . $[any\ n]/C$ selects n intervals of C . $[any\ -n]/C$ selects all but n intervals of C . The any slicing operator has a subtly different operation when used in a declarative statement — in that case, it refers to an interval without specifying which one. This usage of any has a close relationship to the existential quantifier of the predicate calculus.

D. Examples of Collections

Table 1 gives a list of English phrases and their corresponding expressions in the collection representation.

IV APPLICATIONS

The reason for constructing this representation is to provide a framework for a scheduling system. The previous sections have shown how terms commonly used in scheduling can be easily expressed. The representation was designed to address three areas of our group's research on scheduling: graphical display, natural language translation (primarily generation), and reasoning (about schedules).

The illustrations in Figures 1 and 2 indicate the type of graphical display that would be generated by the system. The definition of each calendar can be made to contain simple graphical display information, such as the shape and orientation of any "boxes" in which they and their contents are shown. The boxes in Figures 1 and 2 are

unlabelled. An interval of a calendar could also carry tags that could be used to label the boxes or to organize the data in a tabular form.

The bus schedule of Figure 3 provides a good illustration of this. The schedule is constructed as an order 2 collection, where each interval has been tagged. The collection prefers to display intervals with the same tag in the same column. The intervals in turn prefer to display only their start times. Notice that in several places a table entry is blank. Despite this, displaying the table presents no problem.

Hampshire	Amherst	UMass	Smith	Mt. Holyoke
—	—	8:20	8:35	8:45
10:00	10:10	10:20	10:35	10:45
11:00	11:10	11:20	11:35	11:45
... Every hour ...				
6:00	6:10	6:20	6:35	6:45
7:00	7:10	7:20	—	7:45
8:00	8:10	8:20	8:35	—
11:00	11:10	11:20	11:35	11:45

Figure 3. A Bus Schedule

The appointment calendar display of Figure 4 would be treated in a similar fashion. In this case, the collection of appointments is superimposed on another collection to provide the time grid, with the roles of tags and starting times reversed in the displayed table.

The English text in Table 1 indicates the type of natural language that could be produced or processed by the system. Expressions in the collection representation can be almost literally translated into natural language with comprehensible results. Similarly, statements can be eas-

Table 1.	English	Collection Representation
	Mondays	$2/Days$:during: <i>Weeks</i>
	Januaries	$1/Months$:during: <i>Years</i>
	First Monday in January 1986	$1/Mondays$:during: <i>Januaries</i> :during: 1986/ <i>Years</i>
	or equivalently:	$1/(2/Days$:during: <i>Weeks</i>) :during: $1/Months$:during: 1986/ <i>Years</i>
	First of every month	$1/Days$:during: <i>Months</i>
	First Monday of every month	$1/Mondays$:during: <i>Months</i>
	Last two Mondays of every month	$[-1, -2]/Mondays$:during: <i>Months</i>
	Week of the 15th of each month	$the/Weeks$.overlaps. $15/Days$:during: <i>Months</i>
	First full week of each month	$1/Weeks$:during: <i>Months</i>
	Week of the first of the month	$1/Weeks$.overlaps. $1/Days$:during: <i>Months</i>
	First week of the month	$1/Weeks$.overlaps. <i>Months</i>
	U.S. Election Day	$1/Tuesdays$.>. $1/Mondays$:during: <i>November</i>
	The first (or only) day of t	$1/Days$.overlaps. t
	The day after t	$1/Days$.>meets. $-1/Days$.overlaps. t
	Any day of the week	$any/Days$:during: <i>Weeks</i>
	Any day this week	$any/Days$:during: <i>Weeks</i> .overlaps. (<i>Today</i>)

	:00	:20	:40
9	Bart	Bob	
10	Diana		
11	Egon		Robin
12	Lunch		

Figure 4. An Appointment Calendar

ily translated since the temporal components of the statement are not distributed across a number of quantifiers and predicates. For example, the statement

«Roy-worked» contains *Weekend-Days* :during: (*January*)

can be glossed as "The time that Roy worked included the weekend days in January."

Since the expressions are stored symbolically, the system need only generate the actual intervals that it needs. For example, for the expression

23/Seconds :during: *4570/Minutes* :during: *1986/Years*

the system naturally would not generate a data structure containing the 31536000 seconds in 1986 before selecting the one desired. If the system was asked whether two expressions conflicted and could not determine this by purely symbolic means, it still would not need to generate all the intervals in each collection. Only those subcollections and intervals that have been determined to be possible candidates for conflicts need to be generated (and this process can be done recursively).

If scheduling conflicts occur, the system can replace specific slicing operators with the *any* operator. For example, the system could make the following successive generalizations in searching for a non-conflicting schedule:

the/Mondays :during: *1/Weeks* :during: *Months*
the/Anyday :during: *1/Weeks* :during: *Months*
the/Mondays :during: *any/Weeks* :during: *Months*
the/Anyday :during: *any/Weeks* :during: *Months*

Our motivation for this work has been to provide a framework for the scheduling system. We are in the process of building a scheduling system around the representation. We believe that the consideration of collections of intervals is essential to the scheduling domain and that the notation and accompanying semantics introduced in this paper provide a natural medium for that consideration.

ACKNOWLEDGMENTS

We would like to thank Scott Anderson, Carol Broverman, John Brolio, David Lewis, James Pustejovsky, Penelope Sibun, Philip Werner, Mary-Anne Wolf, and Bev Woolf for their assistance in this research and/or the prepara-

tion of this paper. We would also like to thank Peter Ladkin for sending us advance copies of his papers presented at this conference.

APPENDIX

The intersection of two intervals is defined by:

$$t \cap u \equiv (\max(t_\alpha, u_\alpha), \min(t_\beta, u_\beta))$$

The cover of two intervals is defined by:

$$t \cup u \equiv (\min(t_\alpha, u_\alpha), \max(t_\beta, u_\beta))$$

The union of two intervals ($t \cup u$) is defined only if the intervals overlap or meet and is equal to the cover of the two intervals. The empty interval $\epsilon = (\infty, -\infty)$ and any interval that has $\alpha \geq \beta$ is automatically replaced by ϵ . This definition is motivated by the desire to have $t \cap \epsilon = \epsilon$ and $t \cup \epsilon = t$, for any t .

We use the following binary relations on intervals:

$$\begin{aligned} t \text{ overlaps } u &\equiv t \cap u \neq \epsilon \\ t \text{ during } u &\equiv (t_\alpha \geq u_\alpha) \wedge (t_\beta \leq u_\beta) \\ t \text{ contains } u &\equiv u \text{ during } t \\ t < u &\equiv t_\beta \leq u_\alpha \\ t > u &\equiv t_\alpha \geq u_\beta \\ t \leq u &\equiv (t_\alpha \leq u_\alpha) \wedge (t_\beta \leq u_\beta) \\ t \geq u &\equiv (t_\alpha \geq u_\alpha) \wedge (t_\beta \geq u_\beta) \\ t \text{ meets } u &\equiv (t_\beta = u_\alpha) \end{aligned}$$

The during, \leq and \geq operations form partial orders. Note that $t \leq u$ is not equivalent to $(t < u) \vee (t = u)$; however, $t < u$ is equivalent to $(t \leq u) \wedge \neg(t \text{ overlaps } u)$.

REFERENCES

- Allen, James F., 1985, "Maintaining Knowledge about Temporal Intervals" in Brachman and Levesque, *Readings in Knowledge Representation*. Morgan Kaufmann, pp. 509-521.
- van Benthem, J.F.A.K., 1983, *The Logic of Time*. D. Reidel, Boston.
- Ladkin, Peter, 1985, "Comments on the Representation of Time" in Proceedings of the 1985 Distributed Artificial Intelligence Workshop, Sea Ranch, California, pp. 137-156.
- Ladkin, Peter, 1986a, "Primitives and Units for Time Specification" in the Proceedings of the National Conference on Artificial Intelligence, Philadelphia, Pennsylvania.
- Ladkin, Peter, 1986b, "Time Representation: A Taxonomy of Interval Relations" in the Proceedings of the National Conference on Artificial Intelligence, Philadelphia, Pennsylvania.
- Rescher, Nicholas, and Urquhart, Alasdair, 1971, *Temporal Logic*. Springer-Verlag, New York.

A Plan-Oriented Approach to Intelligent Interface Design

Carol A. Broverman
Karen E. Huff
Victor R. Lesser

COINS Department
Graduate Research Center
University of Massachusetts/Amherst
Amherst, MA 01003

Intelligent Interfaces

In complex domains of computer-based professional work, there is a need for intelligent interfaces which assist practitioners (as opposed to novices) with sequences of actions which meet desired goals and are consistent in their global context. It is not a question of replacing the practitioner with an expert system, but rather of cooperatively supporting the work of the practitioner with an intelligent assistant. This assistant would bridge the gap between the practitioner's perspective on problem-solving activities and the computer system's perspective on tool invocations and resource usage.

Using predefined, hierarchical activity definitions (call them plans), the intelligent interface can monitor the conversation between user and computer system, recognizing the commands issued as instantiations of the plans, or automatically generating primitive commands for a high-level plan requested by the user. Such an intelligent interface would combine facilities for plan recognition and plan execution, with a embedded planner used to extend the predefined plans as needed. The assistance provided to users would include:

- detecting actual and potential errors, including errors of global strategy
- recovering from and correcting errors using context and goal information
- creating and managing agendas of work to be performed, by predicting specific future actions based upon past actions

- summarizing accomplishments of terminal sessions, partitioning activities into domain-oriented tasks
- automatically performing steps in a plan or completing a plan, thus shifting from a mode in which the user has the initiative to a mode in which the interface assumes control.

Realizing this type of intelligent interface requires various AI techniques because:

- during recognition, the information needed for a definitive interpretation of a command may not be complete. The interface will have to generate selected alternatives, make choices, and be prepared to retract the interpretation at a later time.
- the predefined plans may be under-specified; for example, information might be lacking to distinguish which of several higher level plans a given plan will be part of. Further, definition of a plan may be approximate, and based in part on heuristic knowledge.
- It will usually be too burdensome to write predefined plans to cover all possible situations, so there will be a need to generate new plans dynamically (either to interpret user actions or to carry out some high-level plan). An embedded planner will add robustness to the interface.
- the practitioner must make decisions in the course of problem-solving, and that reasoning must be modeled in order to be "checked". Even though it appears unrealistic at present to model all decisions, declarative knowledge and automated reasoning techniques can be used for selected subdomains.

Relationship to Power Tools

We believe that the intelligent assistant approach and the power tool approach are complementary. The key to integrating them lies in uniform definition, representation, and management of knowledge. Both intelligent interfaces and power tools depend upon appropriate knowledge at appropriate levels of abstraction. Sharing selected knowledge makes improved user-support possible.

Intelligent interfaces can pass contextual constraints to the tools, describing the problem-solving situation in which the tool is invoked. And the tools can pass descriptive information back to the intelligent interface, describing characteristics of domain entities which specify and/or constrain their future use. In fact, the integration of intelligent interfaces and power tools can be carried out at multiple levels. When a power tool becomes sufficiently complex, an intelligent assistant needs to be embedded within the power tool to assist the user in selecting sequences of tool facilities to carry out desired goals.

Implementing Intelligent Interfaces

We have implemented an intelligent interface called POISE [1,2,3], which does simple plan recognition. It is integrated with DEC FMS office automation facilities and VMS mail facilities; a demonstration system is now running. POISE uses hierarchical plans whose definitions are behavioral (event-driven) -- lower-level plans are combined via grammar rules with temporal operators to define higher-level plans. A semantic database is used to describe all objects being manipulated by the plans.

The POISE system provides solutions to the problem of incomplete information through its focusing mechanisms (which limit the generation of alternatives to certain plans "in focus") and through truth maintenance (to retract wrong interpretations and re-interpret). It also provides predictive capability through static and dynamic constraint propagation, handling both constraints from the semantic database and from the temporal plans.

A Second Generation System

We are currently designing and implementing a successor to POISE called GRAPPLE (Goal Recognition And Planning Environment). In POISE, there are limitations on the kinds of reasoning which can be performed about the relationships among plans. We have found that behavioral plan definitions, which lack explicit goals and preconditions, are not sufficient for a planner to synthesize new plans. Without explicit goals, there is no way to specify that an action may be omitted because its goal has already been met. Nor is there enough knowledge to recognize failure of a plan -- failure recovery must be built into the plan rules, making the rules rather complex, and preventing special reasoning about failure and recovery.

In GRAPPLE, each plan has an explicit goal, a precondition and a set of effects (state-changes in the semantic database, some of which may be conditional). All plans have internal constraints used for specialization and internal consistency checking. Composition of plans is specified by enumerating the appropriate subgoals (without making any choices of specific plans to achieve those subgoals). The ordering of plans is determined dynamically by preconditions being met. Thus, GRAPPLE plan definitions follow the classical state-driven formalism, with the current state recorded in the semantic database. In addition to this fundamental change in the nature of the plan definitions, we have extended the plans to include "offline" actions, so that we can model user decision-making and the user's beliefs about the state of the work.

In the GRAPPLE formalism, plan failure is recognized when the goal of the plan is still false after the effects are posted. Ordering information is given on a per plan basis, and need not be given for sequences of plans since this can be calculated from the precondition information. If a plan has a subgoal which is already satisfied, then no plan need be executed to achieve the subgoal. Finally, new plans can be added without changing existing plans, because decomposition is through subgoals, not through other plans directly.

Incorporating First Principles Knowledge

As we have worked with plan definitions of either the state-driven or event-driven type, we have recognized that there is additional knowledge about the domain which is not appropriately expressed in the plans themselves. This is particularly true in specialized domains such as software development, where there is a rich set of technical concepts (such as versions, history, configurations, properties and bugs of modules) and a broad range of *first principles* knowledge about programming. This knowledge forms a closed-world for reasoning about actions, and will, we believe, be an important addition to the intelligent interface.

This first principles knowledge can be used in the intelligent interface in several different ways, to improve interface performance and extend more assistance to the user. Using the first principles knowledge to generate tentative bindings of plan parameters will result in earlier, more detailed prediction, and will also limit the number of alternatives to consider during recognition or execution of plans. It provides an alternative to simple heuristics such as "prefer plan continuation to start of new plan" for choosing among alternatives, which may be increasingly important as the number of

alternatives grows or when plans are inherently under-specified. It can be used to double-check decisions made by the programmer (modeled in the offline plans). And, first principles knowledge can provide additional semantic distinctions between apparently equivalent actions (fixing a bug versus adding a new feature) so that future programmer decisions (such as what tests to run) can be anticipated and double-checked.

Status

We have defined the GRAPPLE plan and semantic database formalism, and are currently working on plan recognition algorithms, including constraint handling and focusing. A large set of plans for a Unix/C software development environment has been written in the new formalism, and we are starting to formalize the first principles knowledge for this domain. We have also started work on appropriate meta-plans (such as execute-plan, undo-effects, save-plan-state, suspend, redo-failed-plan, etc.) which will, we hope, provide realistic interpretations for the entire spectrum of user actions.

References

- [1] Broverman, C.; Croft, W.B., "A Knowledge-based Approach to Data Management for Intelligent User Interfaces", *Proceedings of VLDB 11*, Stockholm, 96-104, 1985.
- [2] Carver, N.; Lesser, V.; McCue, D., "Focusing in Plan Recognition", *Proceedings of AAAI*, Austin, Texas, 42-48, 1984.
- [3] Croft, W.B.; Lefkowitz, L.S., "Task Support in an Office System", *ACM Transactions on Office Information Systems*, 2: 197-212, 1984.

This work is being supported by the Air Force Systems Command, Rome Air Development Center, Griffiss Air Force Base, New York 13441-5700, and the Air Force Office of Scientific Research, Bolling Air Force Base, District of Columbia 20332, under contract #F30602-85-C-0008.

APPENDIX 5-K

Providing Intelligent Assistance
In Distributed Office Environments

Sergei Nirenburg
Colgate University

Victor Lesser
University of Massachusetts

Published in *Proceedings of the Third National ACM Conference on Office Information Systems* (1986).

This work was supported by the Air Force Systems Command, Rome Air Development Center, Griffiss Air Force Base, New York, 13441-5700, and the Air Force Office of Scientific Research, Bolling AFB, DC 20332 under Contract No. F30602-85-C-0008. This contract supports the Northeast Artificial Intelligence Consortium (NAIC).

PROVIDING INTELLIGENT ASSISTANCE IN DISTRIBUTED OFFICE ENVIRONMENTS

Sergei Nirenburg
Colgate University

Victor Lesser
University of Massachusetts

Abstract. We argue here that a task-centered, an agent-centered and a cognition-oriented perspective are all needed for providing intelligent assistance in distributed office environments. We present the architecture for a system called OFFICE that combines these three perspectives. We illustrate this architecture through an example.

1. Introduction.

An expert system for the distributed office environment where there is cooperative activity of a number of workers must provide guidance to the user about how to prioritize his own tasks so that they are coherent with the goals of the whole system. This scheduling problem is the key issue in effective distributed planning. Component subproblems here include managing resources; equalizing workload distribution; managing goal conflicts; maintaining a proper level of redundancy in task execution and especially in information flow; analysing dependencies in the sets of goals, plans and events, etc. Automation of any of the above tasks clearly involves manipulation of many types of knowledge, both domain and control.

To illustrate the problem of local scheduling that takes into account global coherence, consider an office consisting of an executive, E, and his/her secretary, S. Suppose, E is dictating letters to S, and the telephone rings. S answers, and the call appears to be about a very important shipment, and S is asked to provide some information about it. The scheduling choice here is between continuing with the letters (task T1) and performing the request that came over the phone (task T2). We want our system to consider a number of factors here, including the relative importance of the tasks (say, a number of people may be idle in the company because of the lack of raw materials that are to be shipped), the time limitations (suppose, the information is needed before the end of the business day, and it's already 4 p.m.; also, the estimated time of finding the requested information), personal characteristics of S and E, etc. If the secretary were scheduling purely locally, he/she may prefer to schedule T2, but knowing that E will be detained by her doing so, S may prefer T1 based on global coherence considerations. S's knowledge about personal characteristics of E can also be a factor: if E is very conscious of his/her status and importance, then the decision of scheduling T1 is even more strengthened; if not, and if S has the characteristic of being assertive, T2 may be preferred, after an explanation to E.

In what follows we describe OFFICE, a system that provides intelligent assistance in the office environment. First, we trace its genesis from three research projects in connected fields and discuss its functionality. Second, we describe how an office can be modelled in a distributed computer system such as OFFICE and describe its architecture and the basic processing cycle. Finally, we give an example of OFFICE operation where we concentrate on its reasoning capabilities.

The Task-Oriented Perspective.

Our initial effort in developing an expert system in the office domain is the task support system POISE (Croft and Lefkowitz, 1984). POISE has been designed to support office workers in their problem solving activities through the use of plan recognition and planning. In the plan recognition mode the system obtains messages about certain atomic events (such as tool invocations) and tries to determine into which of typical tasks known to the system this event fits. In this manner POISE is able to monitor the activities in an office, predict some future activity and detect some errors. In the planning mode of operation POISE is supplied with a typical task and its parameters and tries to execute as much of it as possible, based on its knowledge of the task structure and the status of domain objects in a semantic database. POISE's knowledge takes the form of an hierarchy of typical tasks represented with the preconditions necessary for the execution of the tasks plus the statement of the intended goal.

The use of object-oriented knowledge in POISE is, however, limited. It currently does not have knowledge about static relationships among participants (agents) in the plans that it monitors. As a matter of fact, POISE does not in practice distinguish or reason about the agents' roles and the objects in plans. Thus, for instance, it does not have the possibility to understand that an unusual event happened if it gets the message that the president of a company typed a letter (and not a secretary). Therefore it cannot infer that the secretary may have a day off or that a goal must be instantiated of changing workload distribution among the employees.

POISE is also limited in that plan processing operates sequentially, one input message at a time. This is a potential efficiency bottleneck, but also is not a good model of the activities in an office, that are typically distributed. POISE plans are structured so that they in principle allow concurrent execution of subtasks of a task. Straightforward transformation of POISE into a distributed system cannot, however, be performed. Since there is no developed agent-oriented perspective, there is no way in POISE to express a fact such as 'Requests made by the manager of the office have priority over those made by other workers'. There is also no way of talking about seemingly independent tasks being actually parts of a cooperative problem solving situation. This includes the considerations of arbitration of competing claims for limited resources. More information and a better organization of knowledge are needed for scheduling in a distributed environment. The workers in an office are largely responsible for scheduling their own activities. Thus we argue for incorporating an agent-oriented perspective in order to handle scheduling issues.

The Agent-Oriented Perspective.

One of the research areas where we can look for ideas of how to implement the agent-oriented perspective is the field of distributed AI. One of the current approaches there is the study of functionally accurate, cooperative (FA/C) distributed problem solving (Lesser and Corkill, 1981; Lesser and Corkill, 1983; Durfee et al., 1984, 1985). With this approach, a problem is solved in cooperation by a set of semi-autonomous processing nodes (agents) that independently generate partial solutions, communicate them through a network to other nodes, receive messages from other nodes, and modify their hypotheses in accordance with new input. The experience of this group has shown that the control problem is difficult; that the network communication is both difficult and computationally expensive; most importantly, it was found that the key to global coherence is in having sophisticated agents who can reason about their own view of processing as well as the views of other agents, including the metaknowledge involved in controlling own processing as well as other agents' processing.

The office world offers a number of additional challenges, since it is very knowledge-intensive. The number of various types of objects is large. The complexity of these objects, both physical (e.g., a computer or a person) and mental (e.g., the role of a secretary in the office), is significant. The same is true as regards the processes that are typically performed in an office. There are also many types of static relationships among the objects (e.g., the one between any two office workers) and the processes (e.g., bidding is a component of purchasing). The requirements of the domain can necessitate changes in the basic network and node architecture of the FA/C problem solving approach.

The Cognition-Oriented Perspective.

The FA/C distributed problem solving concentrated on the architecture of the network and the nodes, with the view of organizing the control structure. The types of knowledge necessary for control and communication in OFFICE are studied in the field of cognitive agency research (e.g. Georgeff, 1984, Moore, 1985, but mainly Nirenburg et al., 1985, 1986). The view of the world in this field is that cognitive agents are immersed in a world which is non-monotonic, in the sense that changes in it can be introduced not only because of the activities of a single agent but also through uncontrolled external events. Agents are capable of a variety of cognitive tasks. They can perceive objects and events in the world. They possess a set of goal types and means of achieving goals of these types: plans. They perform goal and plan generation, selection and execution in complex situations in which many goals and plans coexist and compete for the attention of the agent's conscious processor.

The study of the causes of particular choices of goals and plans by the agent (in other words, reasons for scheduling decisions) is the central point of the approach. The knowledge that underlies the reasons for scheduling domain, communication and control actions is claimed to involve such factors as personality traits, and physical and mental states of the agent, in addition to the knowledge about the domain situation and the typical tasks and goals. The general idea is to use all the types of

knowledge discussed in the cognitive agency approach within the architectural framework inspired by the distributed AI research.

2. An Architecture for a Distributed Office System.

We present here, through an example, an architecture for an intelligent assistance system that integrates the task-, agent- and cognition-oriented perspectives.

2.1. Representing an office.

An office is modelled as a network whose nodes are interpreted as office workers and edges, as communication channels. Every node in the network is a complete problem solver. It typically represents a human and computer program working together. Following POISE, OFFICE deals with typical activities in a university-based research project (RP), namely: purchasing equipment, hiring and travel. The types of agents in the RP office include Principal Investigator (PI), Research Associate (RA), Graduate Student (GS), Secretary (S), Vendor (V) and Accountant (A). A typical instance of a project may involve 1 PI, 2 RA's, 6 GS's, 1 S, 3 V's (e.g., DEC, Symbolics and TI) and 2 A's (say, one in Accounts Receivable and one in Personnel).

The knowledge that agents have about roles in the organization, including their own, is illustrated in Figure 1 for a subset of roles in RP. Figure 2 shows the communication channels for the RP office.

Role knowledge: Secretary (S)

- performs paperwork for organization;
- reports to PI (not other members of organization);
- answers phone;
- types documents;
- handles mail;
- maintains all schedules (meetings, deadlines, etc.);
- reminds project members, vendors, etc. about commitments, etc.;
- seldom initiates activity without request;
- protects PI from too much external interference and low-level chores;
- takes shorthand notes during meetings, etc.

Default necessary and sufficient resources for S: a desk; a chair; a telephone; a computer terminal; office supplies; a shared office (a separate office is unusual)

Role Knowledge: Principal Investigator (PI)

- makes decisions concerning management of resources, including hiring, purchasing equipment and firing;
- manages project progress, which involves distributing equipment, distributing agent responsibilities and general supervision and control, including giving assignments, advice and suggestions to RAs, obtaining (oral and written) reports from RAs;

- acts as the spokesperson for RP;
- procures funds for RP;
- signs disbursement vouchers, etc.

Default necessary and sufficient resources for PI: a separate office (shared office is unusual), a desk, chairs, a conference table (optional), a blackboard, a computer terminal, a telephone.

Figure 1. Knowledge about roles in an RP office.

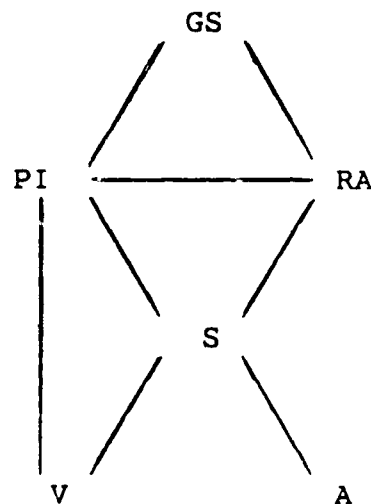


Figure 2. The network of processing nodes in a model of an RP office.

Every agent is aware of its responsibilities to carry out parts of certain plans. They also know who or where from they must seek information that is necessary for them to perform their tasks. *

At any moment t each agent in OFFICE has an agenda of current goals or, more precisely, of current goal instances, as illustrated in (1),

$$\left\{ PU_{\sigma_1}^5, PU_{\sigma_1}^6, HI_{\sigma_3}^1, TR_{\sigma_4}^2 \right\}_t \quad (1)$$

where PU^i , HI^j and TR^k stand for instances of goal types *Purchase*, *Hire* and *Travel*, and σ_i designate subsets of network nodes. Intuitively, at any given moment the office workers are pursuing a number of goals, working in teams. Note that

* Information about the typical agents for all types of tasks is among the knowledge that every agent possesses.

some such goals can be in conflict. Therefore, means of resolving them have to be known to agents.

The architecture of an agent in OFFICE is illustrated in Figure 3. We will not go into the details of knowledge representation here. Suffice it to say that we use a frame-based representation for objects and events. Plans are represented in extended EDL (cf. Nirenburg et al., 1985). An agent has knowledge about its own plans and goals. It also has a representation of the plans and goals of other agents in the network

2.2. How Do the Agents Operate?

A cycle of processing by each agent involves a consecutive invocation of the perceptor, the goal generator, the scheduler, the planner and the executor (cf. Figure 3).

The perceptor

- a) obtains as input *messages* about changes in the world that were received since the previous time cycle (changes are various new states, including results of actions performed by agents in the system). *
- b) 'understands' these actions in terms of *plans* they are parts of and, correspondingly, in terms of what was the *goal* that the agent of that action pursued. This step embodies the *plan recognition* activity of the system, since, in the general case, it must understand plans of others in order to perform its own plan production.

The goal generator

updates the agenda of its goals due to new inputs. Thus, the arrival of new input (2)

(message-14 (2)
(instance-of message)
(speech-act order)
(sender PI-1)
(receiver Secretary-33)
(proposition (communicate Secretary-33
Vendor-101
'what is the price of desk-22?'
Phone))

will lead to the generation of the low-level goal instance 'Get-Info-34' that will be fulfilled when the secretary knows the price of the desk. Even the plan for reaching this goal is specified in the message: using the telephone. This inferencing activity can help the secretary to predict the future steps in working

* Input messages are classified according to their *speech act* character. Messages can be either assertions or requests. Assertions can be definitions, opinions, facts, promises, threats and advice. Requests can be questions (request-info) or commands (request-action). Questions can be either yes/no or wh. Commands are orders, suggestions or pleas. This classification is needed to improve the understanding capabilities of the system (as compared, e.g., with POISE).

toward the higher-level Purchasing goal. Thus, in a spare moment may be devoted to preparing a purchase-order form, even though the time for that in the plan has not yet come.

The scheduler

selects a goal to pursue from among a number of candidate goals on the agenda. An important feature of the system will be the use of knowledge about goal interactions (including conflicts) in scheduling goals for processing. The knowledge that the scheduler uses includes:

- a) the bias factor, determined by the knowledge about authority distribution in the office, that estimates the importance of the agent's decision from the global point of view; thus, a PI's goal is more 'important' than a GS's goal;
- b) information about the level of a goal: whether it is a top-level one or the subgoal of a goal for which another agent is responsible;
- c) predefined relative 'importance' of a goal (purchasing may be given a priority over hiring, for example);
- d) information about the *set* of goals currently on agenda; if a goal facilitates fulfillment of other goals on agenda, its rating grows.

The planner

has the task of providing a plan for the achievement of the goal scheduled by the scheduler. If the agent knows of a *canned plan* that typically leads from the current state to the goal state, simply passes the plan to the Executor (see below). If more than one plan can be used to achieve a given goal, the planner selects one of them, based on static evaluation functions and/or lookahead (at present we use the former alternative) for execution.

If there is no canned plan to go from the current state C to the goal state G, then the agent uses the 'rules of the world' to try to find a sequence of applications of standard plans that leads from C through a sequence of intermediate states (partial solutions) I to G.

The knowledge needed by the planner: the list of plans, the goal-plan table, the rating functions for plans and the 'rules of the world'.

The executor

is called after the planner produces a plan for achieving the current goal.* It performs the following sequence of steps:

- a) creates an instance of the chosen plan (if such an instance does not already exist) and lists it under the corresponding goal on the agenda.
- b) checks preconditions of the plan; if preconditions do not hold (the plan is not immediately applicable) then sets precondition states to be (sub)goal states; puts them on the goal agenda (note that one of preconditions is 'to have values for all non-optional parameters') else expands the agenda tree by substituting the current plan by the sequence of its component plans.

* This is a simplification. In reality, planning and execution steps can be interleaved.

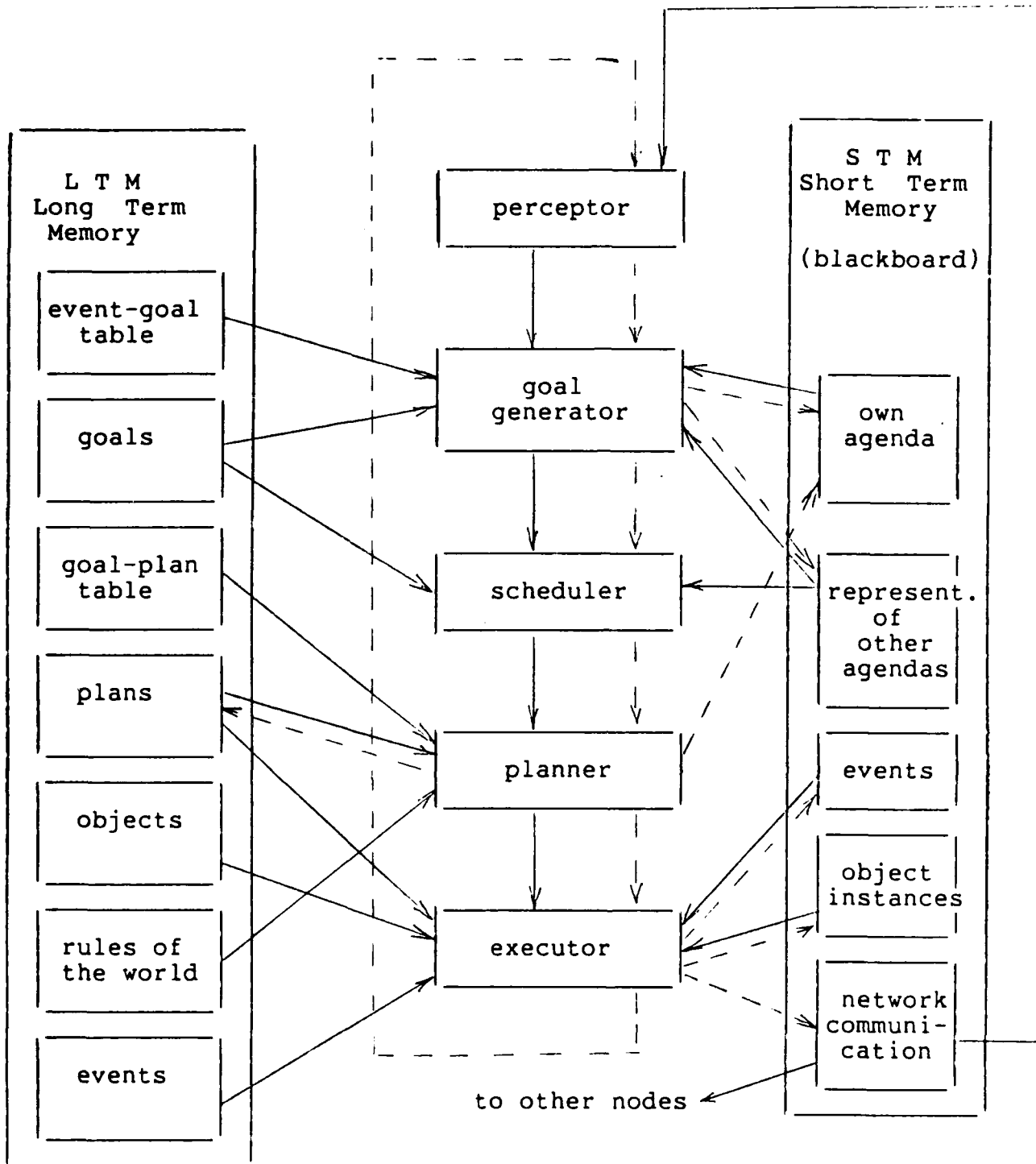


Figure 3. Node Architecture.

—————> flow of data
 - - - - -> flow of control

c) if the first subplan in this sequence has the current node as its agent, it is processed by the executor; if another role in the office is the agent of a subplan, the execution of the current plan is interrupted and a value of its 'status' slot is set to 'suspended' (until other nodes will perform their share of processing).

d) if the plan is 'primitive' the actions specified in it are performed. Then the executor checks whether the plan is completed; if yes, reports this, through the communication channels, to the agent responsible for the next plan in the script.

3. An Example Run of Office.

We will consider 2 top-level goals: PU (purchase) and HI (hire). If agent A has a top level goal G on its agenda, then A is *responsible* for achieving G. If, while performing plan P, A comes across a subplan, SP, in which a different role, B is specified as agent, A must *request* that B perform SP, and *check* whether SP is performed, if a report from B is not received on time.

More than one agent can be responsible for achieving a certain goal: say, both A and B have G on their agendas. In this case A does not request B to perform a subplan but simply *waits* for B to do it. If after N time slices (where N depends on the type of SP) a report from B is not received A sends B a reminder. B exhibits similar behavior.

Each agent learns about other agents' activities from messages coming from those agents or through its own metalevel reasoning based on its beliefs about other agents' goals and knowledge. One of the possible heuristics to restrict the amount of communication among agents is to allow them to report about execution of a certain subplan (SP) only to agent(s) that are (or will be) involved in performing the top level plan that contains SP as a subplan. In other words, not everything that happens in the system is automatically made known to all agents.

We will follow the processing at one node of the network, that of Secretary (S). At the beginning of the run S already has a *nonempty* agenda of plans and goals. It also has a representation of agendas of other nodes in the network. This representation may contain mistakes, because it is mainly a result of plan understanding activities of the node. The contents of S's agenda and S's belief about the agendas of a sample of other nodes at the beginning of our manual trace are given in Figure 4.

S's own agenda:

AGENDA ITEM 1:

PU3 (object = terminal)

communicate (agent = S, destination = PI, object =
[communicate (agent = V1, object = terminal, destination = S)
communicate (agent = V1, object = bill, destination = S)])

check-goods (agent = PI, object = terminal16)

plan-selector (agent = S, object =
[**pay-for-goods** (agent = S, destination = V1, object = bill)
cancel-goods (agent = S, destination = V1, object = (terminal16 bill))])

AGENDA ITEM 2:

process-purch-order5 (object = book)
make-doc (agent = S, doc-type = purch-order, object = book, destination = V2)
communicate (agent = S, object = purch-order, destination = V2)

Secretary's beliefs about PI's agenda:

AGENDA ITEM 1:

PU3 (object = terminal16)
complete-purchase (agent = PI, object = terminal16)

AGENDA ITEM 2:

HI2 (RA)
evaluate (agent = PI, object = candidate3)
make-doc (agent = S, object = offer, destination = candidates)
communicate (agent = S, object = offer, destination = candidates)
select (agent = candidate, object = accept/rej)
make-doc (agent = candidate, object = accept/rej)
communicate (agent = candidate, object = accept/rej)
plan-selector (agent = S, object = [acceptance-track rejection-track])

S's representation of RA1's agenda:

AGENDA ITEM 1:

PU1 (object = book11)
process-purch-order (agent = S, object = book11)
complete-purchase (agent = S, object = book11)

An agenda item consists of the name of a plan and the names of those of its subplans that are not yet executed, with the bindings for their parameters. Plan names are printed in **bold**. Plan names with numbers appended represent plan instances. The above agendas say that the secretary has the plans to facilitate the purchase of a terminal and to facilitate purchasing of a book asked for by a research associate (PU1); S believes PI has plans to hire a research associate (HI2) and to facilitate the purchase of a terminal (PU3). S also believes that RA1 has the plan of purchasing a book (PU1). PI is responsible for both of the plans on its agenda; S is co-responsible for the PU3 plan. In contrast, S is responsible only for a subplan of the top-level plan PU1. RA1 is responsible for PU1.

Figure 4. Sample Contents of the Agendas of an Agent.

Now let us trace the operation of OFFICE through a number of time slices starting with the above state, observing the decision S makes and the changes to its agenda due to new inputs.

----- time slice 1 -----

Suppose, there is one message posted on S's blackboard : message19 from RA2, of type *order*, that asks to get a price for a desk from vendor V by phone. This message is perceived by S and a new goal, GET-INFO11, is generated and put on agenda. S also updates its representation of RA2's agenda by adding there the (inferred) plan of buying a desk. Since S is not responsible for this inferred plan, it does not copy the inferred PU goal to its own agenda.

Next, the scheduler must choose one of the 3 goals on the agenda (PU3 PROCESS-PURCH-ORDERS and GET-INFO11) for immediate processing.

In our example GET-INFO11 will be chosen. This happens because PU3 is out of contention since it is in the stage of waiting for ordered goods (terminal) to come, so the choice is between P-P-O and GET-INFO. P-P-O has, of course, been on agenda for a longer time, but GET-INFO can be performed by just placing a phone call, while P-P-O requires typing out a form. There is no rush on the book order, so the goal that can potentially be achieved sooner is selected. This is an informal statement of one of the *policies* that guide metaprocessing in OFFICE.

Next, a plan *get-info* is found for achieving the chosen goal; this plan is instantiated and the executor runs its first subplan: *communicate15* (agent = S, object = message34, proposition = message19.proposition, destination = V2, type = question, instrument = phone). As a result of that subplan, the vendor is informed about the question.

----- time slice 2 -----

New inputs: a) Message20: a terminal and a bill arrived from vendor V1 b) Message 21: the price for the book arrived from V2.

The messages are perceived and understood as the execution of specific plans traced on S's agenda: a) refers to the two *communicate* plans that are objects of the next component of the plan chosen for the PU3 goal; b) is the response to message19 above.

The above messages do not lead to the generation of any new goals. The scheduler now has the following choice: PU3, P-P-O5 and GET-INFO11. P-P-O5 has the same status as at the previous cycle. PU3 is now at a point where the PI must be told that preconditions hold now for the execution of the *check-goods* plan (because the terminal arrived). Only one action remains to be performed in GET-INFO11, and that is to relay the information obtained from V2 to RA2.

At this point GET-INFO11 is chosen for the following reasons. S knows that PI is currently in a meeting with a candidate for hiring. Even though the importance of the *check-goods* plan is high (in terms of the amount of subsequent processing of the goal), it cannot be performed for the time being and should be rated low. With the other two goals, other agents must wait until S finishes with these plans in order to continue their processing. But GET-INFO11 is closer to completion. Therefore, it is chosen, and S sends the plan (*communicate* agent= S, Destination = RA2,

Object = Message21.proposition) to the executor.

After this plan is executed, the whole GET-INFO11 is deleted from the agenda.

4. Summary and Status.

We hope we have shown that in order to provide assistance in distributed office environments we need to integrate the three perspectives. It is important to carefully choose the task and delineate the world corresponding to it. It is equally important to provide an architecture that can support sophisticated scheduling activities by nodes in a distributed problem solving network. At the same time one should try to explore the sources of real-world knowledge that is used as the basis for scheduling. In addition to the observable world situation the scheduling algorithm must have access to the knowledge about the internal states of the processors, or, in other words, the 'personal profile' of the agents to whom the system provides assistance.

The node-level knowledge and processors have been implemented in Zetalisp on a Symbolics 3600 Lisp Machine. We are currently developing the network level of the system.

References

- Corkill, D.D., 1982. A Framework for Organizational Self-Design in Distributed Problem Solving Networks. Ph.D. Dissertation, University of Massachusetts, Amherst. (Available as COINS Technical Report 82-33.)
- Croft, B.W. and L.S.Lefkowitz, 1984. Task support in an office system. *ACM Transactions on Office Information Systems*, Vol. 2, 197-212.
- Durfee, E.H., D.D. Corkill and V.R. Lesser, 1984. Distributing a distributed problem solving network simulator. COINS Internal Memo, University of Massachusetts.
- Durfee, E.H., D.D. Corkill and V.R. Lesser, 1985. Increasing coherence in a distributed problem solving network. Proceedings of Ninth IJCAI, Los Angeles, August 1985, 1025 - 1030.
- Georgeff, M., 1984. A theory of action for multiagent planning. Proceedings of AAAI-84, 121 - 125.
- Lesser, V.R., D.D.Corkill, 1981. Functionally accurate, cooperative distributive systems. *IEEE Transactions on Man, Systems and Cybernetics*, SMC-11, 81-96.
- Lesser, V.R., D.D.Corkill, 1983. The distributive vehicle monitoring testbed: a tool for investigating distributed problem solving networks. *AI Magazine*, 4, 15-33.
- Nirenburg, I., S. Nirenburg and J. Reynolds, 1985. POPLAR: Toward a Testbed for Cognitive Modelling. Technical Report COSC7, Colgate University.
- Nirenburg, S., I.Nirenburg and J. Reynolds, 1986. Studying the Cognitive Agent. Technical Report COSC9, Colgate University.

APPENDIX 5-L

The GRAPPLE Plan Formalism

Karen E. Huff and Victor R. Lesser

COINS Technical Report 87-08

April, 1987

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

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

Table of Contents

1.0 Introduction

- 1.1 Requirements
- 1.2 Guide to Organization of Report

2.0 Formal Definition of GPF

- 2.1 Overview of Operator Definitions
- 2.2 Major Operator Clauses
 - 2.2.1 Goal Clause
 - 2.2.2 Effects Clause
- 2.3 The Semantic Database
 - 2.3.1 Modeling the Semantic Database
 - 2.3.2 Formalizing the Semantic Database
 - 2.3.3 Relationship Between SDB and Effects Clause
- 2.4 Decomposition Clause
 - 2.4.1 GPF Plan Networks
 - 2.4.2 Style of Decomposition
 - 2.4.3 Final Subgoals
 - 2.4.4 Iterated Subgoals
- 2.5 Constraints
 - 2.5.1 Underconstrained Operators
 - 2.5.2 Special Use of Constraints with Iterated Subgoals
- 2.6 Other Features
 - 2.6.1 On-line versus Off-line Operators
 - 2.6.2 Interface to Real-world Observations
 - 2.6.3 Protection Intervals
 - 2.6.4 Operator Libraries
- 2.7 Use of Predicate Calculus
 - 2.7.1 Evaluation of Plan Formulas
 - 2.7.2 Role of Bindings in Recognition and Execution
 - 2.7.3 Interpretations and Multiple Database States
 - 2.7.4 Construction Interpretations

3.0 Extensions to GPF

- 3.1 Decomposition
- 3.2 Ordering and Forced Execution
- 3.3 Semantic Database Extensions
- 3.4 Specialization Hierarchies
- 3.5 Improved Notation
- 3.6 Specifying Operator Costs
- 3.7 Non-atomic Primitive Actions
- 3.8 Declaration of Variable Names

4.0 Review and Conclusions

- 4.1 How Requirements were Met
- 4.2 Relationship to Other Plan Formalisms
- 4.3 Acknowledgments

5.0 References

Appendix A: Formal Grammar for Operator Definitions

Appendix B: An Operator Library

List of Figures and Tables

- Figure 1: Architecture for an Intelligent Assistant
 - Figure 2: A Basic Operator
 - Figure 3: Static Precondition Examples
 - Figure 4: Blocks World
 - Figure 5: Semantic Database Formalized
 - Figure 6: Extended Blocks World Blocks
 - Figure 7: Primitive Operations for Extended Blocks World
 - Figure 8: Extended Blocks World Structures
 - Figure 9: Operators for Building Towers
 - Figure 10: Expanding a GPF Plan Network
 - Figure 11: Pathological Plan Net Expansion
 - Figure 12: Pathology Resolved
 - Figure 13: Net Expansions with Non-Final Subgoals
 - Figure 14: Iterated Subgoal Examples
 - Figure 15: Successful Realization of Make-tower
 - Figure 16: Use of Constraints
 - Figure 17: Constraints in Iterated Subgoals
 - Figure 18: Use of the Observe Clause
 - Figure 19: A Recognition Scenario
 - Figure 20: Time Line
 - Figure 21: State/Time Diagram
 - Figure 22: Operator Definitions by Specialization
- Table 1: Achievers for Subgoals and Preconditions

1.0 Introduction

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

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

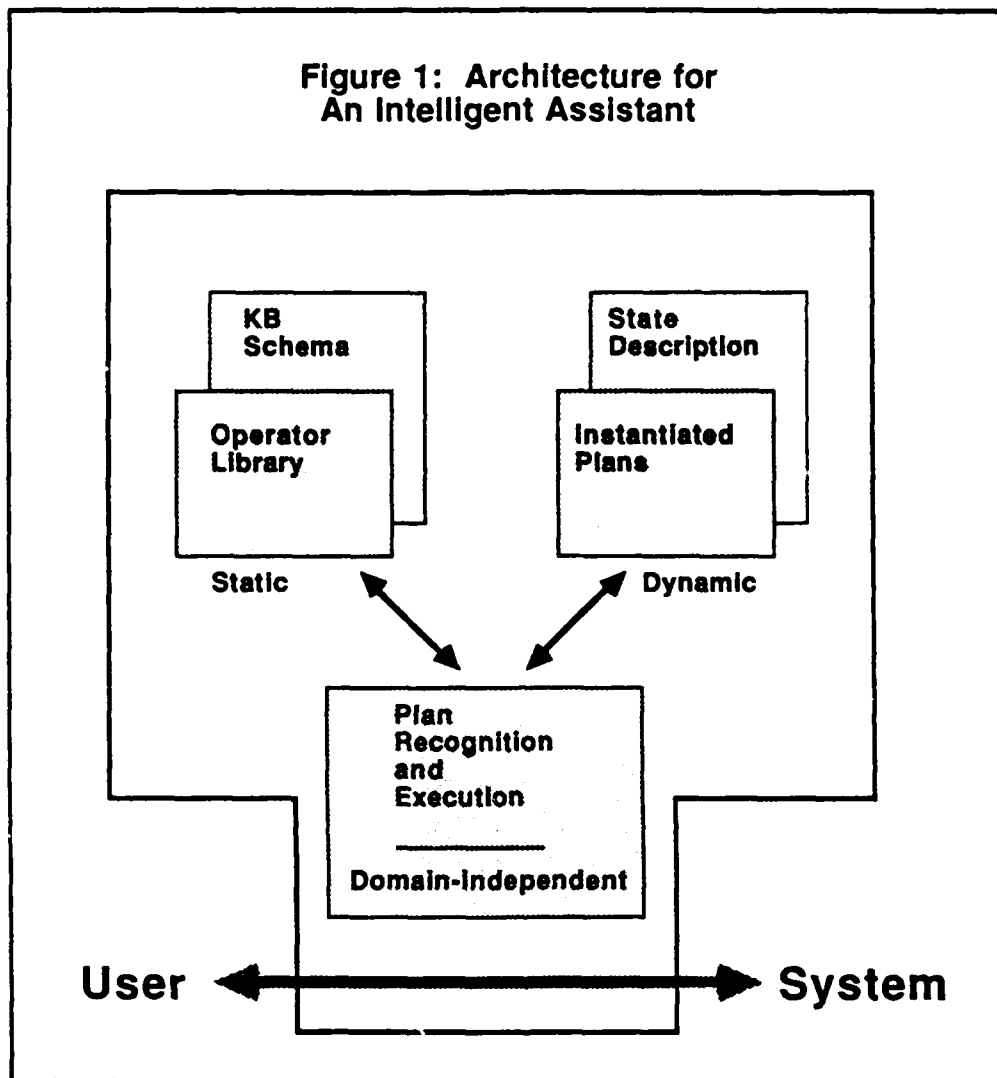
A schematic architecture for such an assistant is given in Figure 1. The assistant itself is domain-independent. Its domain knowledge is embodied in a set of operators (written in the GRAPPLE Plan Formalism GPF) which describe the actions possible in the domain. Using these operator definitions, complete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

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

1.1 Requirements

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

Figure 1: Architecture for An Intelligent Assistant



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

user, it appears that the intelligent assistant augments the facilities of the dumb agent, in effect giving the dumb agent an acceptable level of "smarts".

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

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

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

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

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

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

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

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

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

1.2 Guide to Organization of this Report

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

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

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

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

2.0 Formal Definition of GPF

2.1 Overview of Operator Definitions

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

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

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

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

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

FIGURE 2: A Basic Operator

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

```
(GOAL          on(x,y) )

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

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

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

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

2.2 Major Operator Clauses

2.2.1 Goal Clause

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

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

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

2.2.2 Effects Clause

The effects clause defines a state transition (from an initial state to a final state) which occurs as a result of executing this operator. If execution of the operator was successful, then the goal will be true in the final state of this transition. The effects clause is expressed as a set of atomic

database operations such as making predicates true or false or creating new database objects; taken together as a single, uninterrupted database transaction, these atomic operations define the state transition. In the case of a complex operator, the state transition takes place after all subgoals are achieved.

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

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

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

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

of a operator failing. In this case, the database operation is conditional upon feedback from the real world. Examples of these two uses of conditional effects are as follows:

- Suppose we assume that when we stack one block x on another block y , x can come either from the table or from on top of another block z . Then, in the stack operator, the "side-effects" involving the new status of x will be either deleting $onable(x)$ or deleting $on(x,z)$. The choice of "side-effect" is conditional on the status of x prior to the stack action.
- Suppose the child's robot could not lift blocks whose weight was greater than a certain threshold, and suppose further that the only way to gauge the weight of a block is to attempt to stack or unstack it. Then it is possible that a stack operation would fail if the block were too heavy. In this case, either the effects as given in Figure 2 would be achieved (along with an effect recording that the weight of the block is below the threshold) or the blocks would remain in their original position and we would achieve a single effect: namely, recording that the weight of the block is above the threshold.

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

- If we were building structures and had operators to paint blocks, then we might have a operator to make a structure red, consisting of painting all blocks in the structure red. The blocks are marked red (in the SDB) as each of the subgoals is accomplished, and the structure is marked red (in the SDB) as an effect of the paint-structure-red operator.
- If we are building a vertical structure with three blocks, then we can do so with two stack operations. The effects of the stack operations show x on y and z on x . As effects of the vertical-structure operator, we can introduce an object representing the structure, and show that blocks x , y and z are part of it. These effects give a higher-level semantic interpretation to the world state than is possible with just $on(x,y)$ and $on(z,x)$.

2.2.3 Precondition Clause

The precondition clause establishes constraints on the initial state in which operator execution can start; these constraints must be met in order for the state transition defined in the effects

clause to be valid. Another way to look at the precondition clause is to say that it defines an appropriate start state from which the goal may be achieved via this operator.

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

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

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

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

(PRECOND (A , B AND C))

the complete normal precondition is understood to be the formula:

A AND B AND C

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

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

FIGURE 3: Static Precondition Examples

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

(GOAL          on(x,y) )

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

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

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

(GOAL          ontable(x) )

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

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

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

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

2.3 The Semantic Database

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

2.3.1 Modeling the Semantic Database

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

FIGURE 4: Blocks World

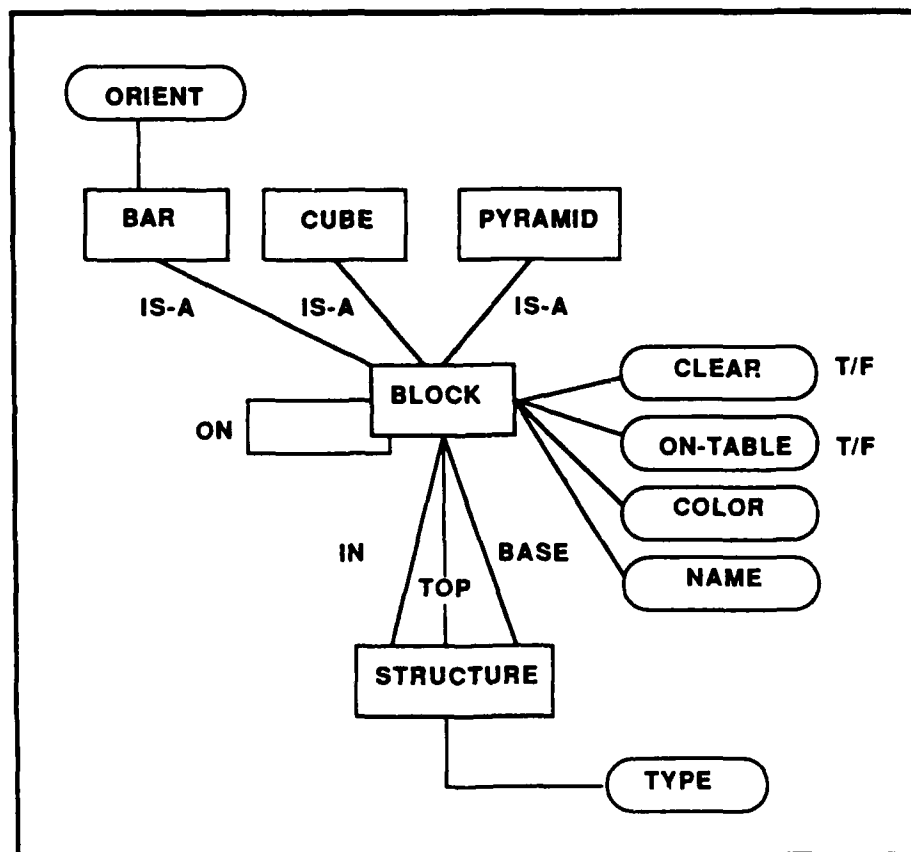


FIGURE 5: Semantic Database Formalized

Predicates (Extensional) attributes)

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

Functions

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

Constraints

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

Predicates (Extensional, from ER

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

Predicates (Intensional)

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

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

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

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

Figure 6: Extended Blocks World Blocks

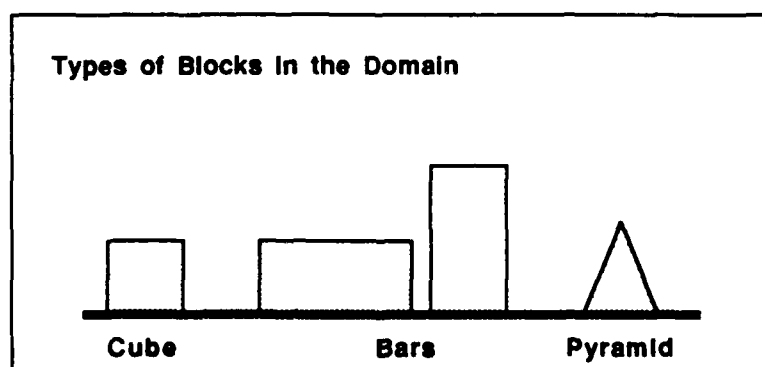


FIGURE 7: Primitive Operators for Extended Blocks World

(OPERATOR start-struct IS-PRIMITIVE

; this is an operator for moving a single block on top of another, thereby starting a new
; structure.

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

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

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

(OPERATOR extend-struct IS-PRIMITIVE

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

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

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

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

(OPERATOR remove-from-struct IS-PRIMITIVE

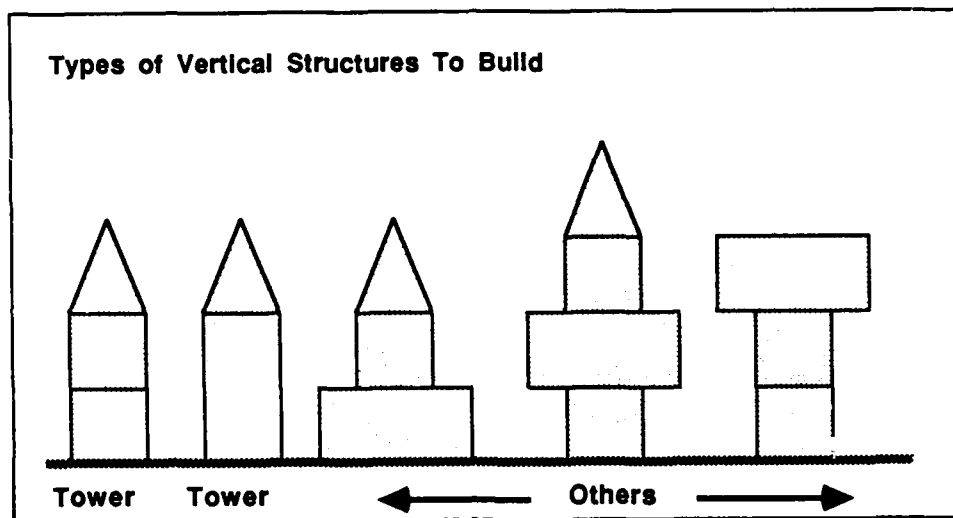
; this is the basic operator for taking one block out of a structure. If there were only two
; blocks in the structure, we disband the structure.

(GOAL NOT committed(x))

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

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

Figure 8: Extended Blocks World Structures



2.3.2 Formalizing the Semantic Database

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

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

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

- Define arbitrary predicates with n arguments to represent interesting formulas with n variables composed from the foregoing (extensional) predicates and functions, with qualifiers as needed. We call these predicates intensional, because they are not directly recorded in the database; their truth/falsity can be *computed* from the facts recorded in the semantic database.
- Predefined predicates are provided for basic relationships on attribute values (strings, integers, and booleans): equal is overloaded for all attribute value types. The usual substring, greater-than, less-than, etc. can also be used. These predicates are also intensional.

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

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

- The "law of gravity" constraint of Figure 5 states that for all blocks, either they rest on the table or they are supported by another block, but not both. If the operator *start-struct* had the error of omitting the (*DELETE ontable(x)*) from the Effects, then such an error would be caught by testing the law of gravity constraint on the state after a *start-struct* action.
- Suppose our domain included an entity type E with at least two different attributes (A1 and A2), and we had separate plans which set these attributes. Suppose we had a constraint *FORALL x: greater-than(A1(x),A2(x))*. During recognition, we may see an action involving A1 with a particular value VA, but not know for certain which entity was involved. If A2(E1) is greater than VA, then we can rule out E1 right now, thus using the constraint to reduce the number of possible interpretations. But if A2(E1) is not set, E1 is a valid possibility. Suppose we choose to guess E1. Later, we see an action setting A2, and suppose we know for certain that entity E1 is involved. Now suppose that this leads to a state where, for E1, A2 is greater than A1. We must have made a mistake in guessing E1 on the action setting A1.

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

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

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

2.3.3 Relationship between SDB and Effects Clause

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

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

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

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

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

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

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

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

2.3.3.2 ADD/DELETE The ADD and DELETE operations take as arguments an (extensional) predicate or a conditional (extensional) predicate. (Limitation to extensional

predicates is necessary because those are the ones explicitly recorded in the database -- the intensional predicates are computed from the extensional ones , and thus can be used for querying but not updating the database). Thus, their form is:

ADD/DELETE <extensional predicate>

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

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

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

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

2.3.3.3 SET The SET operation has the form:

SET <attr spec>

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

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

of possible values. (Relaxation of this restriction is discussed in the Section 3.3.2.) An example of an unconditional SET operation appears in all the plans in Figure 7.

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

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

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

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

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

2.4 Decomposition Clause

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

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

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

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

FIGURE 9: Operators for Building Towers

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

(GOAL tower(s))
(PRECOND (TRUE))
(DECOMP (FINAL SUBGOAL build-foundation
(in(s,x) AND base(s,y) AND on(x,y))
(FINAL SUBGOAL add-pyramid
(in (s,z) AND on(z,x))
(CONSTRAINTS (type-block(y,cube)) ; base-is-cube
(type-block(x,cube)) ; middle-is-cube
(type-block(z,pyramid)) ; pyramid-at-top
(EFFECTS (SET (type-struct s tower))))

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

(GOAL tower(s))
(PRECOND (TRUE))
(DECOMP (SUBGOAL build-it
(in(s,x) AND base(s,y) AND on(x,y)))
(CONSTRAINTS (type-block(y,bar)) ; base-is-bar
(orient(y,vert)); bar-is-vertical
(type-block(x,pyramid)) ; pyramid-at-top
(EFFECTS (SET (type-struct s tower))))

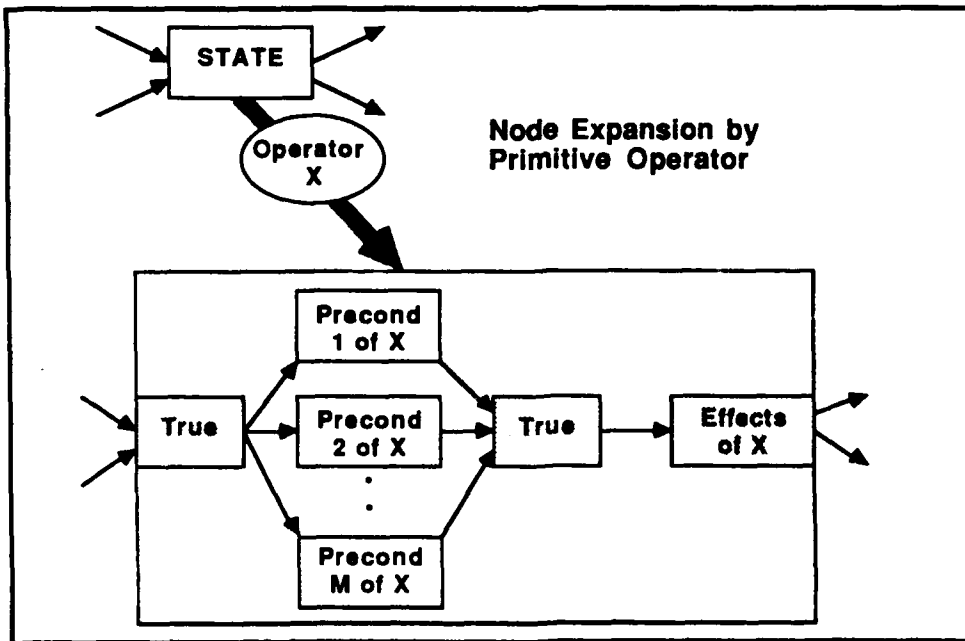
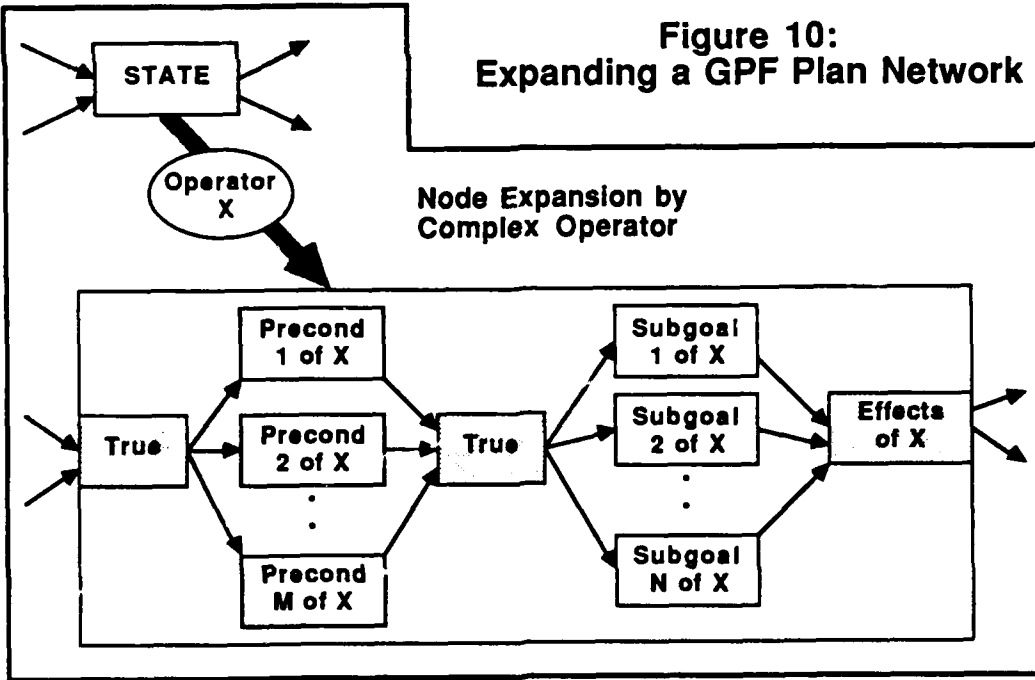
2.4.1 GPF Plan Nets

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

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

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

Figure 10:
Expanding a GPF Plan Network



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

2.4.2 Style of Decomposition

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

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

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

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

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

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

2.4.3 Final Subgoals

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

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

Figure 11:
Pathological Plan Net
Expansion

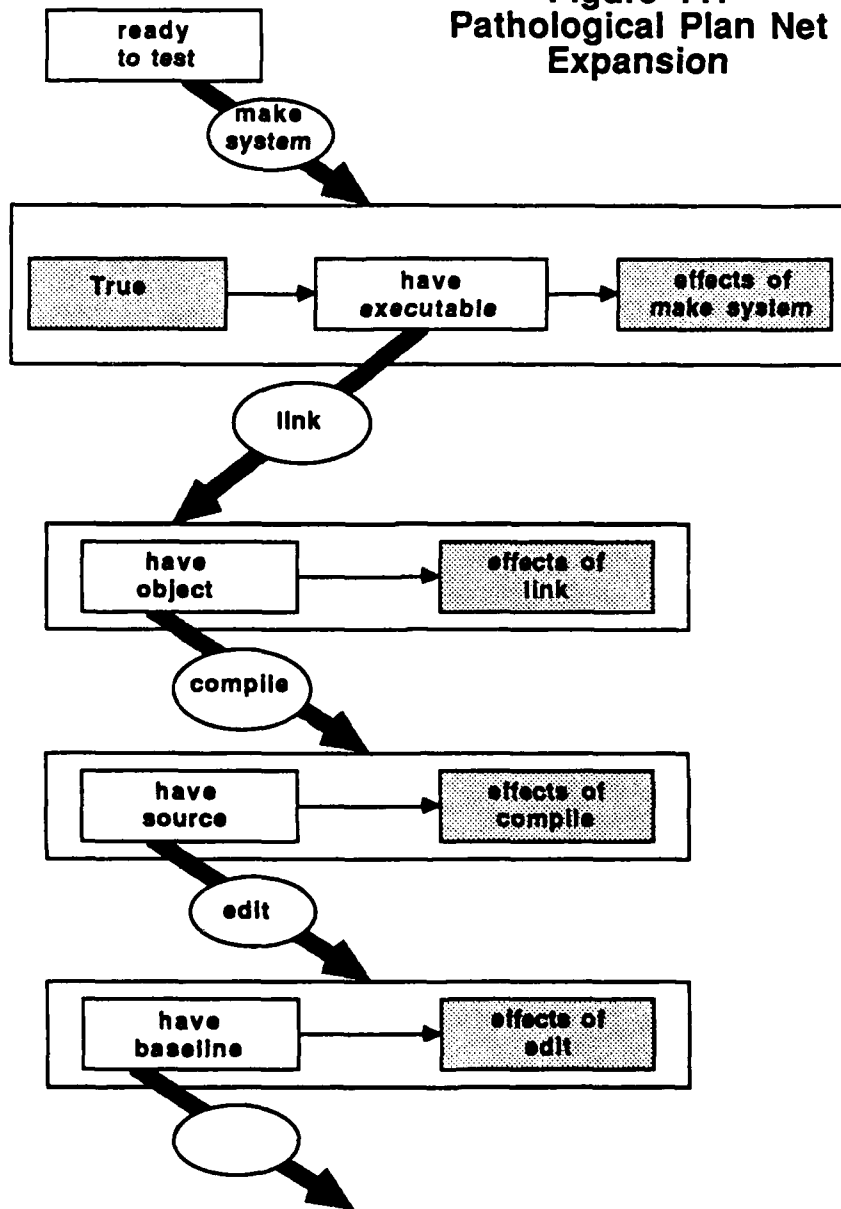
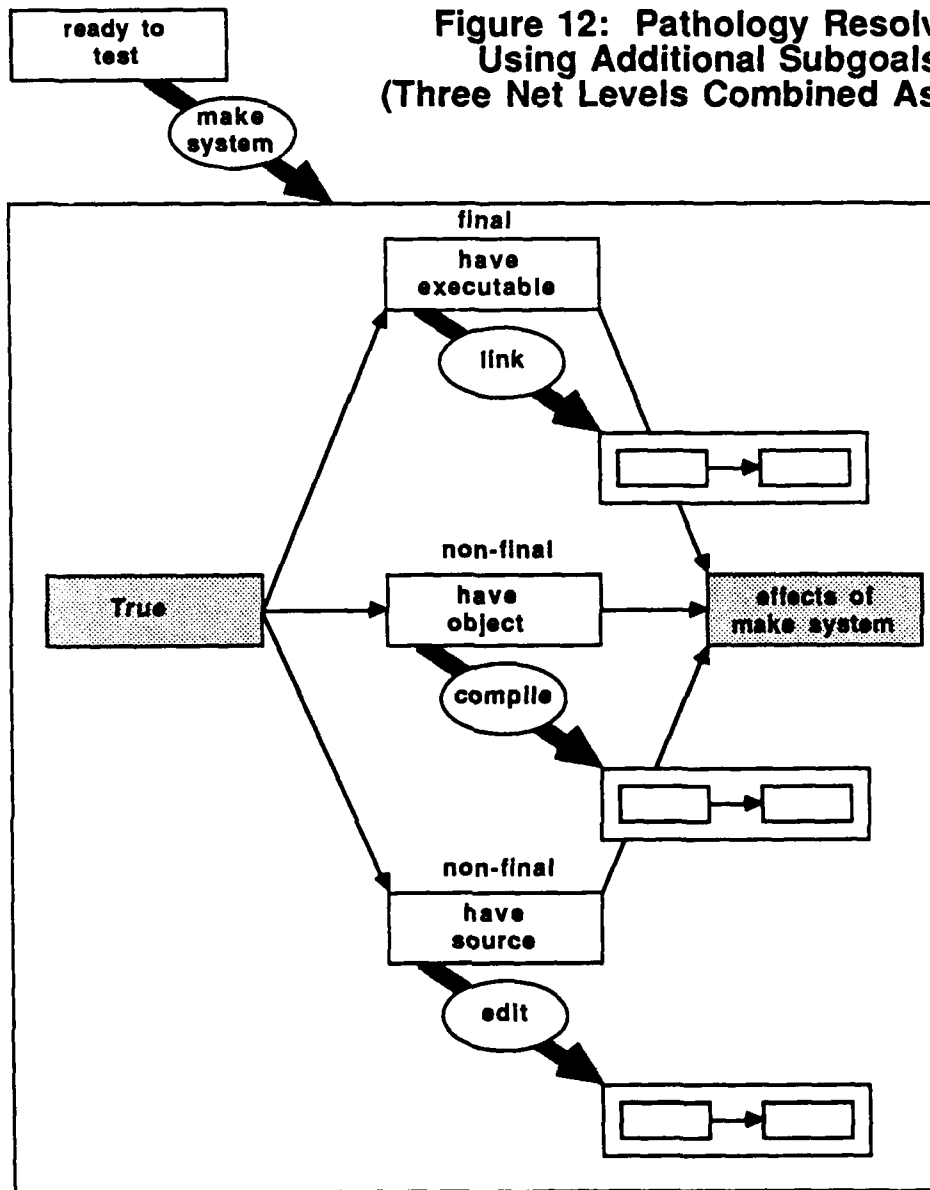


Figure 12: Pathology Resolved Using Additional Subgoals
 (Three Net Levels Combined As One)



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

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

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

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

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

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

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

Note that none of these subgoals are FINAL -- they all deal with intermediate steps to be passed through. The remove-extraneous-blocks subgoal requires that in the course of constructing the tower, we pass through a state where block x is on the top of the structure. For a tower made from scratch, this subgoal will be satisfied simultaneously with the

build-foundation subgoal, so no additional actions are required. For a tower made by partially dismantling an existing tower (for which build-foundation is already true), this subgoal forces the removal of the extraneous blocks. This subgoal is obviously not a final subgoal, since ultimately z will be placed on x.

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

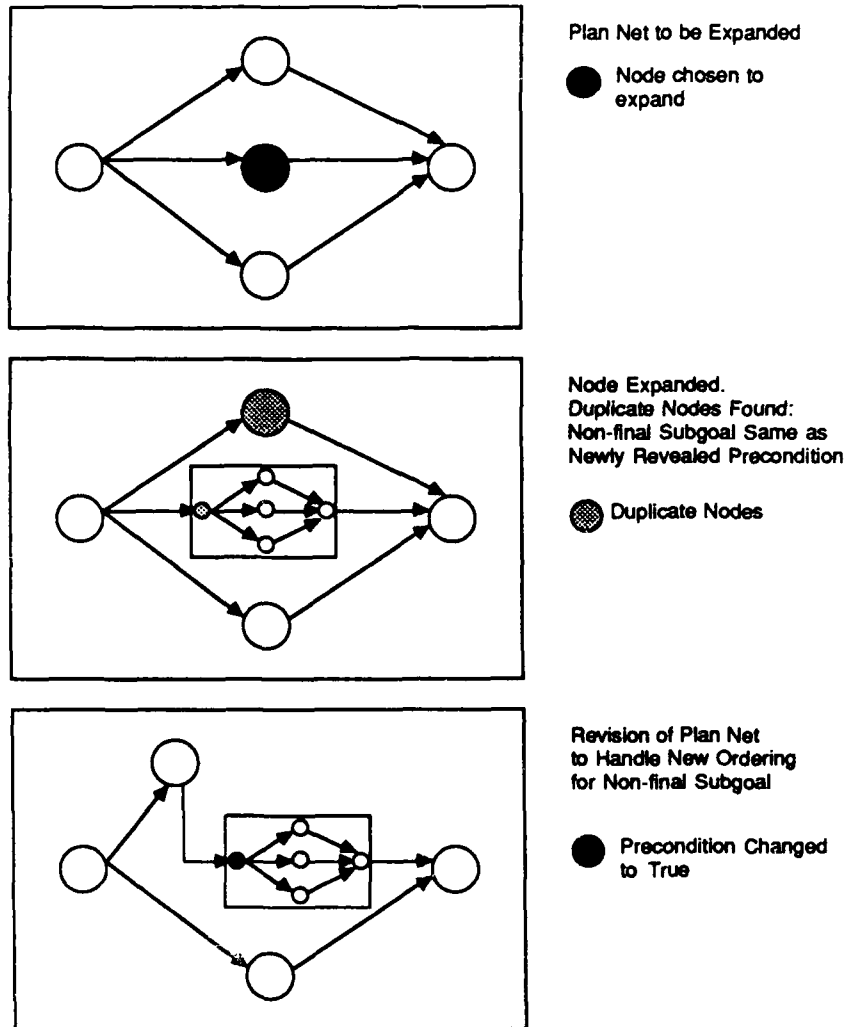
With both FINAL and non-FINAL subgoals all explicit, the operator mirrors the thinking of a person trying to build a tower: "Well, let's see. I've got to make three blocks of the right sort available; that's not simple -- they could be buried deep in existing structures. I've got to stack the two cubes.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

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

2.4.4 Iterated Subgoals

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

Figure 13: Net Expansions with Non-Final Subgoals



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

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

Figure 14: Iterated Subgoal Examples

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

(GOAL tower(s))

(PRECOND (TRUE))

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

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

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

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

(OPERATOR dismantle-struct IS-COMPLEX

(GOAL top(s,x))

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

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

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

(EFFECTS)

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

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

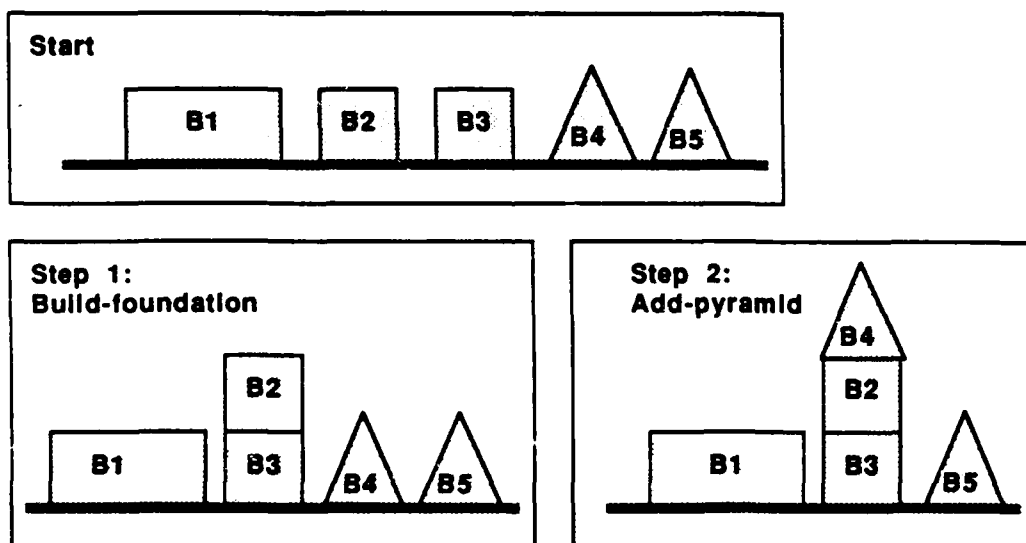
2.5 Constraints Clause

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

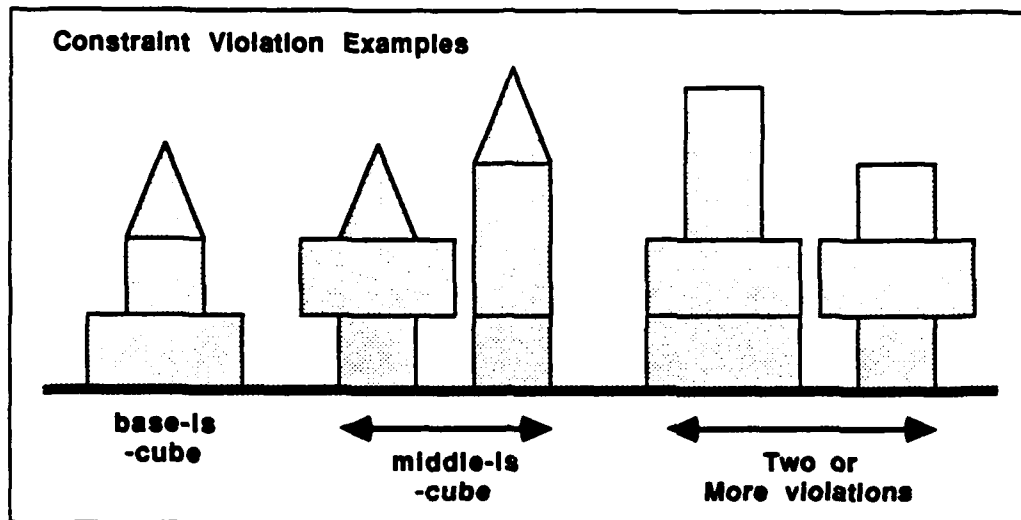
2.5.1 Underconstrained Operators

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

**Figure 15:
Successful Realization of Make-tower**



**Figure 16:
Use of Constraints**



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

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

We call an operator *cooperative* if it uses the *OUT-OF-SCOPE* constraint on one or more

variables. Such an operator cannot be executed automatically without cooperation from the user.

2.5.2 Special Use of Constraints with Iterated Subgoals

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

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

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

Figure 17: Constraints in Iterated Subgoals

(OPERATOR make-arbitrary-sized-tower IS

(GOAL tower(s))

(PRECOND (TRUE))

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

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

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

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

....))

2.6 Other Features

2.6.1 On-line versus Off-line Operators

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

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

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

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

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

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

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

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

If after-the-fact recognition of off-line operators were the only possibility, then the power gained from defining them would be negligible. However, it will commonly be the case that supporting on-line actions (typically information gathering actions) will be taken *before* the off-line action; by recognizing these related actions, it is possible to *predict* the off-line action

(with parameter bindings) before it has occurred rather than to *deduce* it after it has occurred. For example, the decision to use certain blocks in a structure might be dependent upon their distances from one another or their weight (move the lightest, closest blocks in preference to the heaviest, furthest blocks.) Thus, selecting blocks could be represented as a complex operator, with subgoals for measuring and weighing, in addition to a subgoal to make an actual selection. (This last subgoal would be achieved by an off-line operator)

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

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

- to represent key user decisions about variable bindings, whether or not those decisions are OUT-OF-SCOPE. In particular, all uses of the OUT-OF-SCOPE constraint can be placed within off-line operators, but off-line operators are not required to contain an OUT-OF-SCOPE constraint. Example: whether or not the intelligent assistant has enough knowledge to bind choices for which blocks to use in a structure, the decision to use a block can be represented by an off-line operator.
- to represent user decisions to terminate a COMPLETES style iteration when no on-line operator can be used to do so. In this case, the operator which signals completion is an off-line operator. Example: The operator to build a tower of arbitrary height has an appropriate completing on-line operator: to place the pyramid. But, an operator to build a column (defined to be a stack consisting only of cubes) of arbitrary height would have an off-line operator to signal that the last cube has been added.
- to create and initialize domain entities which do not have a physical existence in the real world. In some ways this is the converse of the iteration termination: here an off-line operator is used to set up the circumstances in which another operator can be started, rather than completed. This use of an off-line operator helps to bridge the gap between the modeled and unmodeled parts of the domain, or between the physical reality of the world and the user's interpretation of that reality. Example: a separate off-line operator can be written to create structures, rather than having them be created in start-struct.

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

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

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

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

2.6.2 Interface to Real-world Observations

GPF was designed to be used in an application combining plan recognition, plan execution, and planning. In order to accomplish recognition and execution, an interface to the real world is needed to receive the stream of primitive actions as they occur (recognition) or to generate a

stream of the primitive actions to be performed (execution). This interface is not strictly needed for systems which only do planning (i.e., the dynamic construction of plans in the abstract) because a planner typically deals with idealized executions, rather than actual executions including the possibility of failure.

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

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

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

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

plans. But when effects involve non-local changes as in state changes for failed plans, the STRIPS assumption begins to break down. The approach we have taken is perhaps most appropriate when simplified assumptions about the pervasiveness of failure are used.

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

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

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

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

(OBSERVE (namey namex))

and there would be additional constraints as follows:

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

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

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

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

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

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

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

FIGURE 18: Use of the Observe Clause

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

(GOAL          on(x,y) )

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

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

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

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

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

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

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

The "description" is needed to support a query to the user for the value during plan execution; the variable name is needed in order to have a name by which to refer to the value (for example, in the effects) during plan recognition. An example use for the case of the two-speed robot

described above would be this observe clause for the stack x on y operator:

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

2.6.3 Protection Intervals

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

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

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

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

2.6.4 Operator Libraries

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

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

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

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

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

is achieved by an operator whose goal is B when there is a database constraint of A *IFF* B or *IF* B *THEN* A .

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

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

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

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

For a given library of operators, we can define a set of library conditions, made up of all subgoals of complex operators and all (separate parts of) normal preconditions for both complex and primitive operators. For each library condition, the matching algorithm to

compute the possible achievers can be executed and the achiever information stored for later use in expanding plan hierarchies. The achievers for all library conditions in the operator library of Appendix B are given in Table 1.

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

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

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

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

Table 1: Achievers for Subgoals and Preconditions

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

* Ruled out dynamically when static preconditions and constraints considered

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

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

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

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

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

2.7 Use of Predicate Calculus

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

2.7.1 Evaluating Operator Formulas

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

<u>Plan Formulas</u>	<u>Bindings</u>	<u>Semantic Database</u>
P(x,y)	$\langle x, A1 \rangle$	P(A1, A2)
Q(z)	$\langle y, A2 \rangle$	Q(A3)

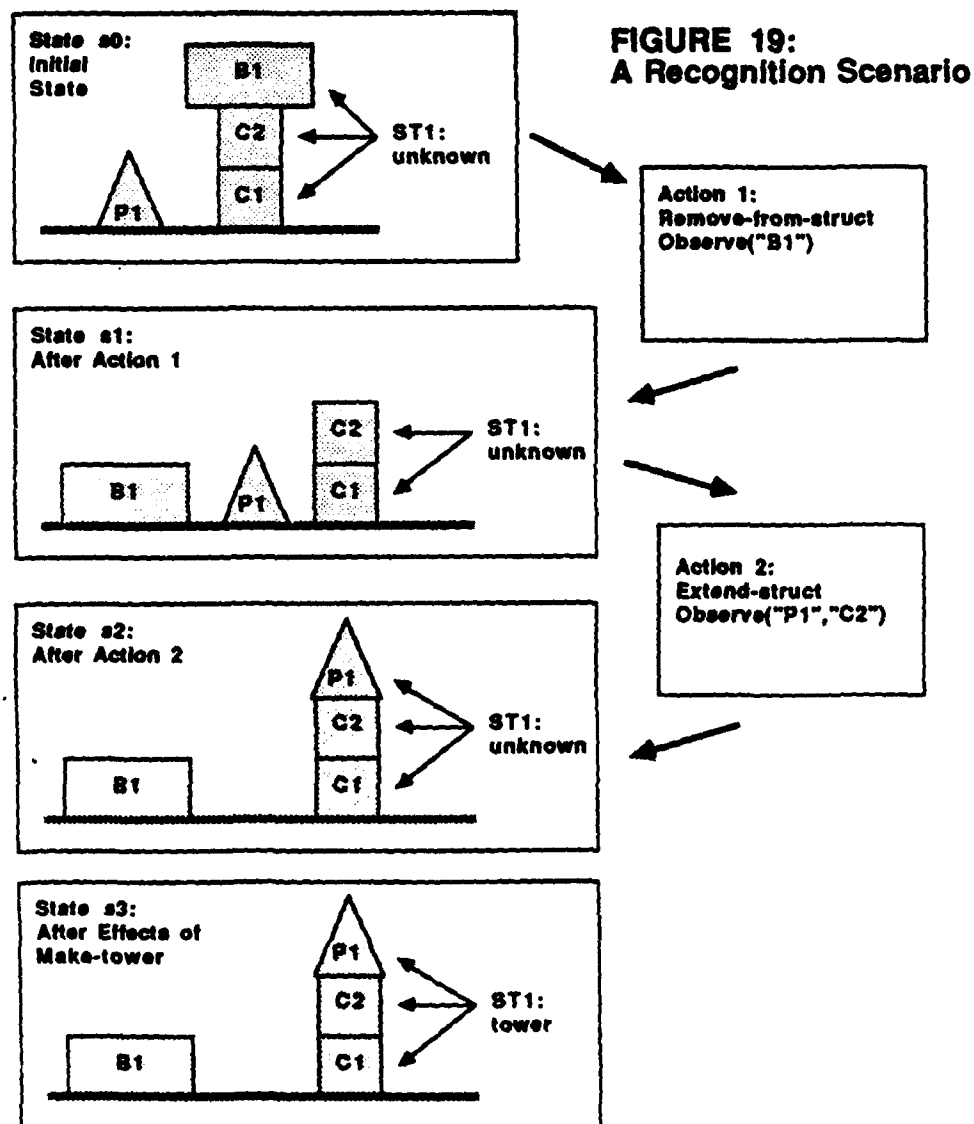
P(x,y) can be evaluated directly since both x and y have bindings; furthermore, given these bindings, P(x,y) is true. In contrast, Q(z) cannot be evaluated as is, because z is without a binding. However, we can evaluate *THERE EXISTS* $z / Q(z)$, which will evaluate to true exactly for the binding $\langle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

2.7.2 Role of Bindings in Recognition and Execution of Plans

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

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

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



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

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

2.7.3 Interpretations and Multiple Database States

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

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

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

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

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

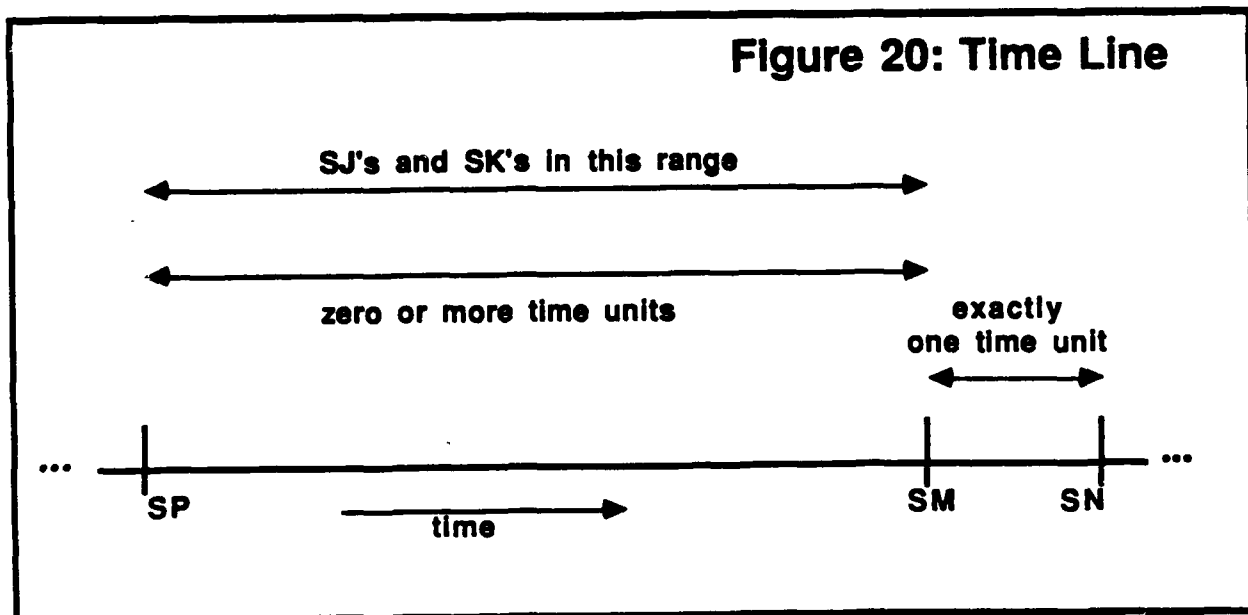
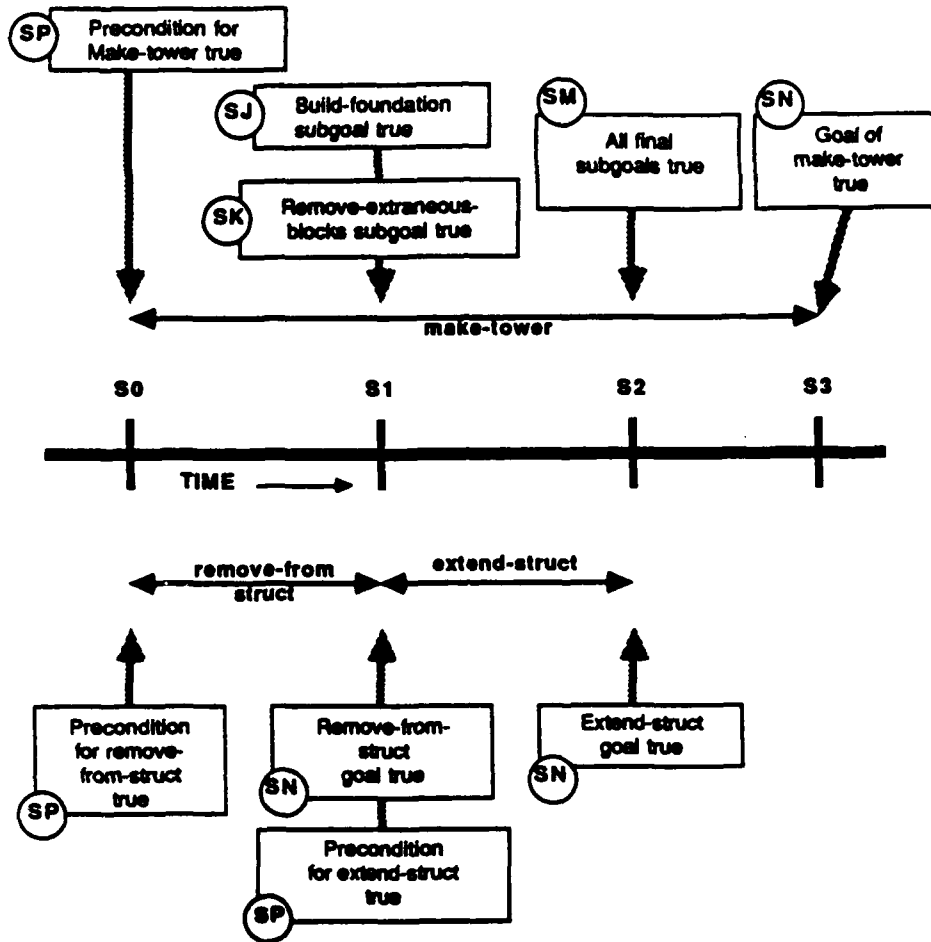


Figure 21: State/Time Diagram



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

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

remove-from-struct

<s,ST1>
<x,B1>
<y,C2>

SP = state s0
SN = state s1

extend-struct

<s,ST1>
<x,P1>
<y,C2>

SP = state s1
SN = state s2

tower

<s,ST1>
<x,C2>
<y,C1>
<z,P1>

SP = state s0
SM = state s2
SN = state s3

SM(build-foundation) = state s0

SM(add-pyramid) = state s3

SJ(remove-extraneous-blocks) = state s2

SJ(make-first-cube-available) = none

SJ(make-second-cube-available) = none

2.7.4 Constructing Interpretations

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

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

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

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

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

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

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

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

b.). Thus, if we have bindings for X we can propagate them to Y, and vice versa, by using these consistency rules.

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

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

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

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

Candidate bindings are a function of the other bindings used in the query. If those bindings should have to be retracted, then the candidate set will have to be reevaluated. If we arrive at a candidate set $\langle x, \{c1\ c2\ c3\} \rangle$ from evaluating $THEREEXISTS\ x\ / p(y,z,x)$ with $\{\langle y,b1 \rangle\ \langle z,b2 \rangle\}$, then if b2 later proves to be the wrong binding for z (because it was itself one of several candidates), the candidate set $\{c1\ c2\ c3\}$ should be thrown out and the query on

predicate p reevaluated. In summary, every candidate binding is contingent upon the bindings used in the query which produced the candidate.

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

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

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

3.0 Extensions To GPF

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

3.1 - Decomposition

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

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

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

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

(ONLY OPERATOR <operator name list>)

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

Following the same form, we could allow:

(PREFERRED OPERATOR <operator name list>)

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

3.2 Ordering and Forced Execution

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

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

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

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

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

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

3.3 Semantic Database Extensions

3.3.1 Macros for Effects Clauses

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

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

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

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

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

3.3.2 Use of Intensional Predicates in ADD/DELETE

GPF operators are precluded from adding and deleting intensional predicates. Thus, we cannot have an effect which includes $ADD committed(x)$. If we were to allow this, then we could have situations where x was committed (because $committed(x)$ is recorded in the database) but

$in(s,x)$ would be false for all structures s (because we had not also recorded what structure x was committed to). That means that the database is logically inconsistent, given the definition of "committed".

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

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

3.4 Specialization Hierarchies

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

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

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

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

Figure 22: Operator Definitions by Specialization

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

(GOAL AND color(s,red))

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

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

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

(GOAL OVERRIDE tower(s))

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

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

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

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

Operator definitions through specialization could be handled entirely by the operator reader (which takes the external operator form presented here and manufactures the definitions in an internal form), so that the use of specialization is entirely transparent to the recognition/execution algorithms. However, this will prevent us from using specialization to solve the problem of the filter program distinguishing between start-struct and extend-struct

(we would have to define these operators as complex ones with a single subgoal satisfiable by the stack primitive operator). In other cases as well, it might be that the existence of specializations might be of some value to the intelligent interface. Expanding the specializations within the operator reader thus is probably not the best way to proceed.

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

3.5 Improved Notation

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

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

3.6 Specifying Operator Costs

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

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

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

3.7 Non-atomic Primitive Actions

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

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

3.8 Declarations of Variable Names

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

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

4.0 Review and Conclusions

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

4.1 How Requirements Were Met

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

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

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

The POISE-related requirements for handling exceptional situations are principally met by introducing goals and preconditions for all operators. Executing an operator is redundant when its goal is already satisfied. Once preconditions are present, the temporal ordering of operators can be deduced. The grammar rules become superfluous under their original interpretation (an ordering to be followed); with their omission, the formalism becomes truly state-based. In any

case, the preconditions allow precise control on operator ordering, while the temporal rules approximated the ordering. Concurrency of actions at any hierarchical level is implicit in the fact that any operator whose precondition is true can be executed. The introduction of a goal for each operator allows operator failure to be detected: an operator fails when its goal is not true after its effects are posted. Modularity of operator libraries is ensured because the subgoal decomposition is through states to be achieved by a choice of other operators, not directly through those other operators.

4.2 Relationship to Other Plan Formalisms

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

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

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

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

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

4.3 Acknowledgments

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

5.0 References

- [1] Broverman, C.A., and W.B. Croft, "A Knowledge-based Approach to Data Management for Intelligent User Interfaces", *Proceedings of Conference for Very Large Databases*, 1985.
- [2] Broverman, C.A., K.E. Huff, and V.R. Lesser, "The Role of Plan Recognition in Intelligent Interface Design", *Proceedings of Conference on Systems, Man and Cybernetics*, IEEE, 1986, pp. 863-868.
- [3] Carver, N., V.R. Lesser and D. McCue, "Focusing in Plan Recognition", *Proceedings of AAAI*, 1984, pp. 42-48.
- [4] Chen, P.P., "The Entity-relationship Model: Toward A Unified View of Data," *ACM Transactions on Database Systems*, vol. 1, no. 1, March 1976, pp. 9-36.
- [5] Croft, W.B., L.S. Lefkowitz, V.R. Lesser and K.E. Huff, "POISE: An Intelligent Interface for Profession-Based Systems", *Conference on Artificial Intelligence*, Oakland, Michigan, 1983.
- [6] Croft, W.B., and L.S. Lefkowitz, "Task Support in an Office System", *ACM Transactions on Office Information Systems*, vol. 2, 1984, pp. 197-212.
- [7] Fikes, R.E., and N.J. Nilsson, "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving", *Artificial Intelligence*, vol. 2, 1971, pp. 189-208.
- [8] Huff, K.E. and V.R. Lesser, "Knowledge-based Command Understanding: An Example for the Software Development Environment", Technical Report 82-6, Department of Computer and Information Sciences, University of Massachusetts, Amherst, 1982.
- [9] Huff, K.E. and V.R. Lesser, "Intelligent Assistance for Programmers Based Upon a Formal Representation of the Process of Programming", Technical Report 86-09, Department of Computer and Information Sciences, University of Massachusetts, Amherst, 1986.
- [10] Newell, A. and H.A. Simon, "GPS: a Program that Simulates Human Thought", in Feigenbaum, E. and J. Feldman eds., *Computers and Thought*, McGraw-Hill, New York, 1963.
- [11] Sacerdoti, E.D., *A Structure for Plans and Behavior*, Elsevier-North Holland, New York, 1977.
- [12] Stefik, M., "Planning with Constraints", *Artificial Intelligence*, vol. 16, 1981, pp. 111-140.
- [13] Stefik, M., "Planning and Meta-planning", *Artificial Intelligence*, vol. 16, 1981, pp. 141-169.

- [14] Tate, A., "Project Planning Using a Hierarchical Non-linear Planner", Dept. of Artificial Intelligence Report 25, Edinburgh University, 1976.
- [15] Tenenberq, J. "Planning with Abstraction", *Proceedings of AAAI*, 1986, pp. 76-80.
- [16] Wilkins, D.E., "Domain-Independent Planning: Representation and Plan Generation", *Artificial Intelligence*, vol. 22, 1984, pp. 269-301.

Appendix A Formal Grammar for Operator Definitions

Grammatical Notes

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

Grammar

- <operator> <- (OPERATOR <operator name>
 {IS_COMPLEX | IS_PRIMITIVE} [OFFLINE]
 <goal clause>
 <preconds clause>
 [<decomp clause> | <observe clause>]
 <constraints clause>
 <effects clause>)
- <goal clause> <- (GOAL <formula>)
- <preconds clause> <- (PRECOND (<formula list>) [(STATIC <formula>)])
- <formula list> <- <formula>, <formula list> | <formula>
- <decomp clause> <- (DECOMP <subgoal deflist>)
- <subgoal deflist> <- <subgoal def >
 | <subgoal def> <subgoal deflist>
- <subgoal def> <- ([FINAL] SUBGOAL < subgoal name >
 [<iteration spec>]
 < formula >)
- <iteration spec> <- ITERATED
 | COMPLETES <subgoal name>
 | ITERATED-OVER (<vbl name> : <set spec>)
 | PAIRED-WITH <subgoal name>
- <set spec> <- <formula>
- <observe clause> <- (OBSERVE (<ob name list>) [(RESPONSE <name
 lis:>)]))

<ob name list>	<-	<vbl name> <vbl name> <ob name list> <user supplied value> <ob name list>
<user supplied value>	<-	USER-SUPPLIED (<descrip>, <vbl name>)
<descrip>	<-	<string>
<name list>	<-	<vbl name> <vbl name> <name list>
<constraints clause>	<-	(CONSTRAINTS <constraint list>) (CONSTRAINTS)
<constraint list>	<-	< constraint> <constraint> <constraint list>
<constraint>	<-	<formula>
<effects clause>	<-	(EFFECTS <effects list>) (EFFECTS)
<effects list>	<-	<effect> <effect> <effects list>
<effect>	<-	(<newobj op> <add-delete op> <attr set op>)
<add-delete op>	<-	ADD DELETE <cond-predicate>
<cond-predicate>	<-	<predicate> IF <formula> THEN <predicate> IF <formula> THEN <predicate> ELSE <predicate>
<newobj op>	<-	NEW < obj name> <type name> [WITH (<with spec>)]
<with spec>	<-	<predicate> <predicate>, <with spec>
<attr set op>	<-	SET <attr spec> SET IF <cond formula> THEN < attr spec > SET IF <cond formula> THEN < attr spec > ELSE < attr spec >
<attr spec>	<-	(<attr name> <vbl name> <value>)
<formula>	<-	(<formula>) OLD (<formula>) <predicate> NOT <predicate> <formula> AND <formula> <formula> OR <formula>

<predicate>	<-	<pred name> (<arg list>) TRUE FALSE
<arg list>	<-	<arg> <arg> , <arg list>
<arg>	<-	<predicate> <vbl name> <qualifier> (<vbl name>) <funct> <value>
<qualifier>	<-	THIS NEXT FIRST FINAL EVERY SOME
<funct>	<-	<funct name> (<arg list>)

Appendix B An Operator Library

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

Extensional: setaside(structure,block)

Intensional: ready(structure,block):
 ready(s,y) IFF setaside(s,y) AND NOT committed(y)

 free(block):
 free(y) IFF FORALL s: NOT setaside(s,y)

 empty(structure):
 empty(s) IFF FORALL b: NOT in(s,b)

Constraints: setaside(s1,b) AND setaside(s2,b) IFF equal(s1,s2)

(OPERATOR start-struct IS-PRIMITIVE

; this is an operator for moving a single block on top of another, to be used
; when the structure being built is currently empty of blocks.

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

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

(OBSERVE (namex,namey))

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

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

(OPERATOR extend-struct IS-PRIMITIVE

; this is an operator for moving a single block on top of another, to be used
; when a structure already has blocks in it.

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

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

(OBSERVE **(namex,namey))**

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

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

(OPERATOR remove-from-struct IS-PRIMITIVE

; this is the basic operator for unstacking, i.e., taking one block out of a
; structure. If there were only two blocks in the structure, we disband
; the structure.

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

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

(OBSERVE **(namex))**

(CONSTRAINTS **(name(x,namex)))**

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

(OPERATOR tower-by-adaptation IS-COMPLEX

**;we make a tower from an existing structure which has a cube at its base
; and a cube on top of the base.**

(GOAL tower(s))

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

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

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

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

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

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

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

(GOAL tower(s))

(PRECOND (TRUE))

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

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

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

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

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

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

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

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

(OPERATOR make-alt-tower IS-COMPLEX

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

(GOAL tower(s))
(PRECOND (TRUE))
(DECOMP (SUBGOAL make-bar-available
ready(s,y))
(SUBGOAL make-pyramid-available
ready(s,x))
(FINAL SUBGOAL build-it
(top(s,x) AND base(s,y) AND on(x,y)))
(CONSTRAINTS (type-block(y,bar)) ; base-is-bar
(orient(y,vert)); bar-is-vertical
(type-block(x,pyramid)) ; pyramid-at-top
(EFFECTS (SET (type-struct s tower))))

(OPERATOR make-new-struct IS-PRIMITIVE OFFLINE

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

(GOAL (empty(s))
(PRECOND (TRUE))
(CONSTRAINTS)
(EFFECTS (NEW s structure))

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

(GOAL (setaside(s,x))

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

(CONSTRAINTS)

(EFFECTS (setaside(s,x)))

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

(GOAL (ready(s,x))

(PRECOND (TRUE))

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

 (FINAL SUBGOAL free
 (NOT committed(x))

(EFFECTS))

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

(GOAL top(s,x)

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

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

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

(EFFECTS))

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

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

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

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

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

(EFFECTS))



MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control, Communications and Intelligence (C³I) activities. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of C³I systems. The areas of technical competence include communications, command and control, battle management, information processing, surveillance sensors, intelligence data collection and handling, solid state sciences, electromagnetics, and propagation, and electronic, maintainability, and compatibility.