

AD-A182 732

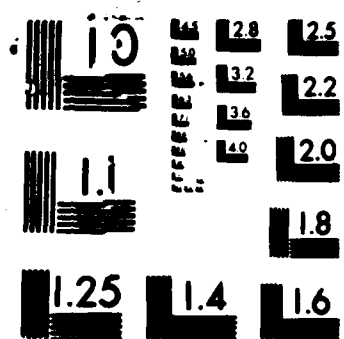
KNOWLEDGE-BASED REPLANNING SYSTEM(U) MITRE CORP BEDFORD 1/1
MA B C DAWSON ET AL MAY 87 RADC-TR-87-68
F19628-86-C-0001

UNCLASSIFIED

F/G 12/5

NL

END
8-87
DTIC



DTIC FILE COPY

12

AD-A182 732

RADC-TR-87-60
Final Technical Report
May 1987



KNOWLEDGE-BASED REPLANNING SYSTEM

The MITRE Corporation

**Bruce C. Dawson, Richard H. Brown, Candice E. Kalish
and Stuart Goldkind**

DTIC
ELECTE
JUL 29 1987
S & D

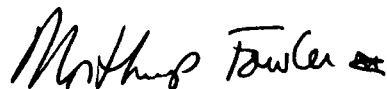
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

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

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-87-60 has been reviewed and approved for publication.

APPROVED:



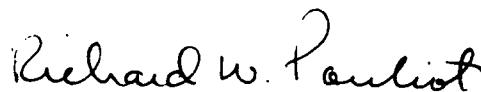
NORTHROP FOWLER III
Project Engineer

APPROVED:



RAYMOND P. URTZ, JR.
Technical Director of Command and Control

FOR THE COMMANDER:



RICHARD W. POULIOT
Directorate of Plans and 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 notices on a specific document requires that it be returned.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE				
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		5 MONITORING ORGANIZATION REPORT NUMBER(S) RADC-TR-87-60		
6a NAME OF PERFORMING ORGANIZATION The MITRE Corporation	6b OFFICE SYMBOL (if applicable)	7a NAME OF MONITORING ORGANIZATION Rome Air Development Center (COES)		
6c ADDRESS (City, State, and ZIP Code) P.O. Box 208 Bedford, MA 01730		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 F19628-86-C-0001		
8c ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700		10 SOURCE OF FUNDING NUMBERS		
		PROGRAM ELEMENT NO 62702F	PROJECT NO 5581	TASK NO 27
		WORK UNIT ACCESSION NO 05		
11 TITLE (Include Security Classification) KNOWLEDGE-BASED REPLANNING SYSTEM				
12 PERSONAL AUTHOR(S) Bruce C. Dawson, Richard H. Brown, Candice E. Kalish, Stuart Goldkind				
13a TYPE OF REPORT Final	13b TIME COVERED FROM Oct 82 TO Sep 85	14 DATE OF REPORT (Year, Month, Day) May 1987	15 PAGE COUNT 88	
16 SUPPLEMENTARY NOTATION N/A				
17 COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)	
FIELD	GROUP	SUB-GROUP		
12	05		Artificial Intelligence Knowledge-based Systems Expert Systems ; Natural Language Understanding Planning Mission Planning	
19 ABSTRACT (Continue on reverse if necessary and identify by block number) The KNOBS Replanning System (KRS) is an extension of the KNOBS offensive counter aircraft mission planning program. KRS uses AI techniques to plan multimission packages and to replan already planned missions. KRS is a constraint-based system which uses a generate and test mode to propose mission plans and to select among them. It contains a strategic, or metaplanning, component based on the theory of Fuzzy Algorithms, which gives the system an ability to decide among plans and to reason about plans.				
20 DISTRIBUTION / AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS		21 ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED		
22a NAME OF RESPONSIBLE INDIVIDUAL NORTHROP FOWLER III		22b TELEPHONE (Include Area Code) (315) 330-7796	22c OFFICE SYMBOL RADC (COES)	

DD FORM 1473, 84 MAR

83 APR edition may be used until exhausted
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

TABLE OF CONTENTS

SECTION 1 - OVERVIEW

1.0 INTRODUCTION	1
1.1 OBJECTIVE	1
1.2 HISTORY	1
1.3 THE PLANNING PROBLEM	2
1.4 TACTICAL MISSION PLANNING	3
1.5 KRS DEVELOPMENT EFFORT	4
1.5.1 KRS in FY83	4
1.5.1.1 Rewrite of KNOBS Code	4
1.5.1.2 Translation of KNOBS Code	5
1.5.1.3 User Interface	5
1.5.1.4 Refueling	6
1.5.2 KRS in FY84	7
1.5.2.1 New Planning Scenarios	7
1.5.2.2 Automatic Planning	9
1.5.3 KRS in FY85	9
1.5.3.1 Interrelated Mission Planning	9
1.5.3.2 Demonstration System	10

Accession For	
NTIS CRA&I	☒
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Date	
Availability Codes	
Distribution	
A-1	

SECTION 2 - THE KRS INTERFACE

2.0 INTRODUCTION	12
2.1 HISTORY OF APE-II PARSER IMPLEMENTATION	12
2.1.1 ATN-Based Natural Language	13
2.1.2 APE-II Development	14
2.2 IMPLEMENTATION OF APE-II	15
2.2.1 APE-II	16
2.2.2 APE-II Dictionary	18

2.2.3	SCRIPTS and User Goals	19
2.3	EVALUATION OF APE-II PERFORMANCE	20
2.3.1	CD Theory is Implementationally Dangerous	21
2.3.2	Do Not Represent Conceptual Dependencies as Trees	24
2.3.3	Reliance on Structural Expectations Carries a Heavy Price	26
2.3.4	Do Not Ignore Syntactic Relationships	26
2.3.5	Believe Not in Primitives: Doomed If You Do	27
2.3.6	Believe in Primitives: Doomed If You Don't	27
2.4	Future Work	28
SECTION 3 - THE KRS ARCHITECTURE		
3.0	KRS Architectural Elements	29
3.1	FRAMES	29
3.1.1	The Data Base	30
3.1.2	FRL Functions	31
3.1.3	Inheritance	31
3.1.4	Procedural Attachment	31
3.2	FRAME INSTANTIATOR	31
3.2.1	Template Slots	31
3.2.2	Automatic Slots and Warning	32
3.3	CHOICE GENERATION AND PLANNING	33
3.3.1	Constraints	33
3.3.1.1	Non-Monotonic Logic	34
3.3.2	Enumeration	34
3.3.3	Ordering	35
3.4	FROM KNOBS TO KRS	36
3.4.1	Constraint References	36
3.4.2	Enumeration	36
3.4.3	Ordering	37

3.4.4	History and Slot Status	37
3.4.5	Multiargument Constraints, Restrictions, Timeliness and Autoplanning	37
3.4.5.1	Possible Solutions	38
3.4.5.2	Implementation	39
3.4.6	Autoplanning	41
3.4.6.1	Problems in Autoplanning	42
3.4.6.2	The New AUTOPLAN Algorithm	42
3.4.6.3	Remaining AUTOPLAN Problems	46
3.4.6.4	Autoplanning the REFUELSVC Slot	46
3.4.6.5	Other Autoplan Changes	47
3.4.7	Resource Tracking	48
SECTION 4 - THE PLANNING COMPONENT		
4.0	INTRODUCTION	50
4.1	WILENSKY'S THEORY OF METAPLANNING	50
4.1.1	MITRE Implementation	52
4.2	STRATEGIES AND FUZZY ALGORITHMS	54
4.2.2	Strategy vs. Tactics	55
4.2.3	Fuzzy Goals	55
4.2.4	Fuzzy Algorithms	55
4.2.5	Plan Schemas	56
4.3	METAPLANNING AND STRATEGIES	57
4.4	KRS IMPLEMENTATION	60
4.4.1	Replanning, Meta-Planning, and Strategy Selection	61
4.4.2	KRS Strategy	61
4.4.2.1	The Declarative Content	61
4.4.2.2	Procedural Implementation	66
4.5	WEAKNESSES OF KRS IMPLEMENTATION OF STRATEGIES	71
SECTION 5 - CONCLUSION		72
BIBLIOGRAPHY		74

LIST OF ILLUSTRATION

Figure		Page
1-1	Refueling Frame Structure	6
2-1	APE-II Dictionary Definitions	17
2-2	CD Representation of "X Threatens Y"	22
2-3	Network Representation	23
2-4	Tree Representation	23
2-5	Definition of "Crash"	25
3-1	AUTOPLAN Algorithm	45

SECTION 1 - OVERVIEW

1.0 INTRODUCTION

1.1 OBJECTIVE

Research into semi-automated planning and reasoning at MITRE started with the development of the KNOBS system. KNOBS was an early effort to unite several capabilities: frame representations for data, rule-based inference, English understanding using a conceptual-dependency parser for question-answering and control, and a constraint-based approach to planning. This complex architecture evolved from a simple rule-based system as a result of the functional and performance demands on the system.

The research described in this report was directed towards expanding the capabilities demonstrated in KNOBS to include replanning through an implementation called the Knowledge Based Replanning System (KRS). The purpose of the system is to create plans for complex resource reallocations, revise those plans when expectation failures occur, and also minimize the impact of any necessary changes on the remainder of the plan. In replanning, the system must deal with previously constructed plans as objects it can reason about and alter; our research has focused on mechanisms to allow for metalevel reasoning and planning. In addition, we have recognized the importance of global strategies to the planning process and explored elements of a system that can use global strategies in its planning processes, understand its own use of strategies, and explain these uses.

To accomplish these goals, we have focused on the development of a declarative formalism for expressing metaplanning knowledge. The formalism developed is able to express global strategies and heuristics along with control information such as subprocess orderings, possible concurrency, and inter-process communication. It may also be able to accommodate widely diverse domains and be interpretable both procedurally and declaratively so that the system can *intelligently explain its own activity*.

The objective of the KRS Project is to explore the application of these techniques to the problem of planning and replanning Air Force Tactical air missions. This report describes the development of KRS during the period FY83 - FY85.

1.2 HISTORY

By the end of FY82, the KNOBS Project had produced a demonstration system which contained a database representing resources, targets, weather, and planned missions in the Central European Theater. The project implemented an appreciable array of software dedicated to the interpretation of tactical knowledge represented through the use of both "frames" and "rules", and consisted of software modules for inference based on the inheritance of properties from the general to the specific, theorem proving via an innovative constraint calculus, and action generation initiated by "demon" processes which observe changes in particular data items and whose effects are determined by a forward chaining production system. KNOBS also contains user oriented interface facilities, including dynamically generated menus and a natural language understanding capability which supports database query, database update, and user redefinition of inference rules.

Quite a wide spectrum of initiative between the planner and the program was represented in KNOBS. The program could, for example, check the completeness and consistency of a plan created by the user, explaining clearly what inconsistencies had arisen, and understanding what had to be rechecked as the planner changed elements of the plan in response to the program's criticism. In this rather passive mode, it could also provide the user with dynamically generated lists of consistent candidates for various plan elements, e.g., the type of aircraft or ordnance.

In a more active mode, the program could rank those same candidates in preference, guided by rules which took into account the current states of both the plan and database representation of the operational environment. These same rules functioned to provide explanations in defense of the system's choices. Finally, in its most assertive mode, the program could construct an entire plan subject to user defined constraints and could detect, and to a limited extent automatically replan, a mission which had become invalid as a result of changing external circumstances.

For all the reasoning power, explanation, and natural language understanding that the original KNOBS program exhibited, it basically planned a single attack mission at a time; in effect, a single line of an Air Tasking Order. It lacked the technology needed to exhibit real-time response to the need of simultaneous replanning a score of missions resulting from changes in goals, resource availability, weather, or targets of opportunity -- what might be characterized as the Combat Operations problem. It also lacked the software needed to perform the hierarchical, elastic replanning required in such circumstances, pervasive enough representations of goals to direct that process, and graphics interfaces to make it comprehensible.

In addition, the KNOBS system lacked the capability needed to support real-time knowledge-based mission monitoring and Air Tasking Order redefinition in response to a changing environment. More profoundly, the classical AI "planning" technology proved to be quite inadequate to the requirements of replanning as it is practiced in the reality of the Combat Operations shop of a TACC. Developing the technology necessary to support monitoring and replanning was the major focus of the KRS project, and the accomplishments of that project is the subject of this report.

1.3 THE PLANNING PROBLEM

The "classical" Artificial Intelligence planning technology may be characterized in general by defining "planning" as discovering an algorithm for the satisfaction of a specified goal, subject to a specified set of constraints. The algorithm is generally defined in terms of the execution of a sequence of operations, chosen from a predefined vocabulary, each with well-defined preconditions and each with well-defined post-conditions, i.e., with a well-defined resulting state-transition. The process of finding the required algorithm is often referred to as "problem solving" and a variety of problem solving techniques have been proposed and studied.

One of the most prevalent of such techniques is referred to as "hierarchical planning". It consists of the decomposition of a problem into a hierarchy of sub-goals in which the instantiation of sub-plans is postponed as long as possible in order to minimize the backtracking required to arrive at an integrated solution. Human problem solving behavior is better approximated by a mixture of top-down and bottom-up processing, often referred to as "opportunistic planning". More specific mechanisms include the "constraint propagation" techniques of MOLGEN as well as other constraint generated solution methods typified by techniques found in automatic programming research. This latter class of problem solving methods is fascinating with respect to replanning as a computer program is, in effect, not a plan, but rather a set of contingency plans, provided in response to the unknowability of certain state descriptions resulting from the unpredictability of program input.

These classical AI planning methods contributed to the eventual knowledge based replanning system, but they were not adequate for bridging the enormous gap between the classical definition of planning presented above and the much more complex exigencies of Combat Operations. The first difficulty was the lack of well-defined goals. Those that exist in the apportionment guidance are often vague, and are certainly subject to frequent change. Typically, the number of satisfactory solutions is very high, with less than optimal results guaranteed by the lack of measures by which to compare plausible solutions. The constraints are subject to continual redefinition, partly as a result of changing tactics, but always as a result of a large volume of changing real or perceived situational truths. Whatever plans exist must be subjected to continual "truth maintenance" challenge as a result of the evolving battle.

Combat Operations is not a "problem"; it is a process. Perhaps the single greatest difference between classical problem solving and the needs of a Combat Operations replanning aid is that the entire AI planning paradigm is often inappropriate in that the support sought by the planning staff need not be automatic problem solution at all. It might be plan "checking" or, as in KNOBS, a wide spectrum of services in which different users or different circumstances demand an altered exchange of initiative between the program and the command staff.

This mixed initiative brings with it a host of problems. These include the need not only to check or generate plans, but the necessity that the program be capable of explaining its advice in terms natural to the users. The most difficult of these problems, one which in fact limits the applicability of a number of AI planning techniques, arises from the possibility of staff intervention in the modification of existing plans. The goal must be "continuity". That is, small changes in circumstances and goals should produce small changes to the pre-existing plan. If that plan, or some part of it, e.g., a particular strike package, was mechanically derived, then one would expect that fact to be preserved along with the actual "object code" of the plan, i.e., the Air Tasking Order (ATO), would be the hierarchical scheme of sub-goals and methods that document the problem solving activity. One would think it would be this planning structure that would be manipulated in order to replan against a changing world. The disaster of mixed initiative is that it allows the user to change the ATO without modifying, or even necessarily understanding, the existing plan structure. This obviously jeopardizes any further mechanical replanning.

A final characterization of Combat Operations that must always be kept in mind is the complexity of the task and the large number of planners working in parallel to ensure the required real-time response. That is, it is neither one problem nor even a process. It is several such processes working in parallel, both in mutual support and in competition for scarce resources. Partial solutions with damaging side effects must be avoided. An eventual automated knowledge-based replanning system must, therefore, be capable of handling a large number of simultaneous, interlocking, automatic and manual planning activities.

1.4 TACTICAL MISSION PLANNING

Tactical air planning is a middle management function, both in terms of the Air Force command structure and in terms of the detail represented in the plan. Organizationally under the Tactical Air Command, mission planning takes place at Tactical Air Control Centers (TACCs). Within NATO, the corresponding facility is referred to as an Allied Tactical Operations Center (ATOC).

The TACC's function is to assign available resources to the various tasks of an "apportionment" order issued by the Joint Task Force Commander. The product of this assignment is an "Air Tasking Order", which summarizes the responsibilities of each unit. For example, Offensive Counter Air (OCA) units are told how many and what kind of aircraft to use, which ordnance they will deliver to each selected target, and at what time. This simplified description ignores all of the TACC's mission monitoring functions, major planning factors like defense suppression and refueling, and details such as the management of radio frequencies.

Automating this planning process should shorten its time cycle - the current 24 hour tasking order generation time is not well suited to most anticipated scenarios. Conventional computer programs could probably go a long way in this direction, and such a program, Computer Aided Force Management System (CAFMS), has been developed [TAC 85]. Its goal is to support a variety of data displays and to aid in the tedious bookkeeping involved in the preparation and distribution of the Air Tasking Order.

Knowledge-based programs, capable of taking on more of the actual planning burden and providing more semantic checks of plan consistency, should be able to accelerate the planning process still further and provide added assurance that the plan is complete and sound.

1.5 KRS DEVELOPMENT EFFORT

KRS was a development which spanned a three-year period from October 1983 to September 1985 (FY83 to FY85). The focus of KRS is replanning and its implementation was built up on a foundation from KNOBS and from lessons learned during the KNOBS development. The starting point for the KRS project was when KNOBS was converted to run on the Symbolics 3600.

1.5.1 KRS in FY83

The major tasks accomplished in FY83 were 1) the improvement of an extensive amount of code to enhance performance and functionality, 2) the rewriting and translation of KNOBS code from Interlisp to Zetalisp so that code would run on a Symbolics 3600 machine, 3) the complete re-implementation of the KNOBS interface to take advantage of the Symbolics high-resolution bit-mapped display and window system, 4) the development of a prototype system capable of exhibiting basic planning of interacting missions by using the refueling example, and 5) supporting RADC with the transition of KNOBS to a contractor for possible implementation of an operational planner called TEMPLAR. The first four items listed above will be discussed in more detail in the following sections.

1.5.1.1. Rewrite of KNOBS Code

It is the evolutionary nature of most computer-based research projects such as KNOBS that code is generated based on the current design to test out ideas that may later be modified. Unfortunately, after several years of development, the system code can become overextended, resulting in badly tangled functional structure and compromised modularity. As development proceeds, each extension to the system adds to its complexity and increases the cost of future extensions, and the law of diminishing returns comes into play: after a point, the effort expended installing changes and keeping the system working exceeds the relatively cheap cost of composing the change.

We had observed this slowly happening with the KNOBS code, and had decided to invest several months overhauling it rather than continue with development. We were at a convenient place to do this in three respects: first, KRS needed a solid foundation as a starting point; second, the TEMPLAR project would soon need the KNOBS source code for its own development, and it would be much easier for contractors to work with KNOBS after it had been cleaned up; third, we were planning to move the entire system from a DEC System-20 (where it was written in Interlisp) to Symbolic Lisp machines and this obviously required that a great deal of code be translated from Interlisp to Zetalisp. We felt that it would be much easier to translate from a clean source system. Also, several known points of inelegance existed in the DEC System-20 version which we did not want propagated to the next version of the code.

In planning the rewrite, the following tasks were agreed upon:

- 1) The interactive frame instantiator would no longer serve as a mechanism for maintaining and switching environments for the constraint interpreter. The instantiator had caused many problems in the past and was beginning to impose a high overhead as switching quickly among frames became important.

- 2) That part of the instantiator which constituted the user interface for the display would be integrated into the display interface.
- 3) The constraint interpreter, which had suffered badly from frequent modifications, was modularized and rewritten. Along with this, the constraint references which appeared in the templates would need to be redesigned to make the specification of a constraint's behavior distinct from that of its domain.
- 4) A list of interface functions to common KNOBS facilities were agreed upon for implementation. The previous interfaces had been in various cases inefficient, inconsistent, nonuniform or nonexistent. These interface functions were considered important because, in a sense, the cleanup of KNOBS was organized around them. Enforcing these conventions had the effect of modularizing the code and isolating the various subsystems so as to diminish complex interactions. It also made possible the separation of many of the facilities from the user interface, so that further development of both the interface and the other facilities could proceed unhindered.

1.5.1.2 Translation of KNOBS Code

Although Interlisp and Zetalisp differ greatly as programming environments, they are actually fairly similar dialects of Lisp. The majority of differences are syntactic; therefore, IZZI (the pattern-based translator provided by Hewlett-Packard) was an appropriate tool. The entire IZZI package consists of three modules: a body of transformation rules, the translator which matches and applies the rules, and a group of auxiliary functions to handle constructs that cannot be translated syntactically (the functions may use the full power of Lisp to examine the context in which the construct is used). Each Interlisp source file was passed through IZZI, producing a corresponding Zetalisp source file, which was then downloaded onto a Lisp machine. Once there, each file was inspected for translation errors, corrected if necessary, and eventually compiled.

Converting KNOBS to Zetalisp took several months. IZZI turned out to be deficient in several respects: its translator was considerably simpler than we had hoped (some general Interlisp forms, such as functions which don't evaluate their arguments, were simply ignored without warning), some of the transformation rules were incorrect, many were missing (IZZI came with two pages of rules; we ended up with ten), and a few of the auxiliary functions mistranslated when the source expression was too complex. Unfortunately, many of these defects were not apparent from examining either the code or the documentation and a few were only discovered far along in the conversion process, while trying to compile the final translated files.

Once the bugs in the target files were found and removed, the remaining code, which could not be processed mechanically, was translated by hand. This had to be done for functions which used machine-dependent I/O, portions of the display interface, the code to support time calculations, etc. Finally, a working version of the KNOBS system began to materialize on the Lisp machines. The primary deficiency of this first version was the simplicity of its user interface, as it did not take advantage of the elaborate window system of the Lisp machine. The next step, then, was to create a sophisticated display interface built on the Symbolics window facility.

1.5.1.3 User Interface

Because of the powerful and sophisticated window system that exists on the Symbolics 3600, and because the implementation of KRS was a departure from the original KNOBS code, it was decided that the original goals for the KNOBS display interface would not be preserved. In other words, no attempt would be made for the new system to run on any terminal other than the Symbolics console.

The KRS display interface as it was re-implemented makes use of most of the facilities provided: pop-up menus, mouse-sensitive items, scrollable dynamic displays, user-definable fonts, multiple-choice menus, and even parallel processes (each mission has a separate process controlling its display). By essentially rewriting the user interface from scratch, we were able to clean up or eliminate many points which had made the KNOBS interface inelegant, inefficient or slow. For example, many mission displays can now occupy the screen simultaneously, and the user may quickly select one by simply pointing to it with the mouse. The user may also "edit" the screen so as to expand a mission display or rearrange the missions on the screen to his liking. Several other features, such as the correct display of multi-valued slots, had become necessary for the development of a refueling prototype.

1.5.1.4 Refueling

The goal of the FY83 work on refueling was to produce a prototype system capable of exhibiting basic capabilities for planning and refueling for Offensive-Counter-Air (OCA) Missions. This is a complex plan of the type we wish to replan. The first step toward a replanning system is, of course, a planning system. We developed a minimal fully automatic refueling demonstration system which could recognize the need for refueling while planning an OCA mission, request refueling for it, finish planning the OCA mission then proceed to plan the refueling. As a "first pass" solution, this system assumes some basic existing refueling capabilities already exist, so that certain more difficult problems may be circumvented.

The refueling world view is based on a structure of frames displayed in Figure 1.1 which were devised to represent the necessary knowledge. The OCA (Offensive-Counter-Air Mission) and RFL (Refueling Mission) frames are the only actual missions in the diagram; the remaining structure may be seen as the conceptual "glue" tying together the various components.

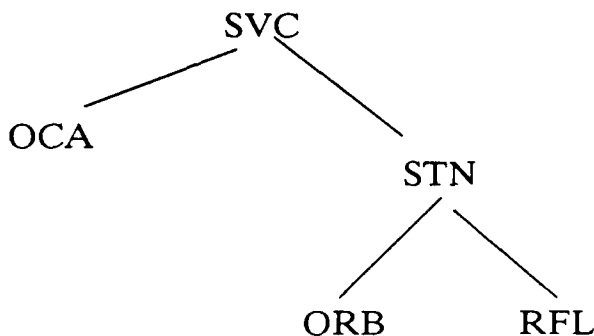


Figure 1.1 Refueling Frame Structure

A refueling mission (RFL frame) is very similar to an OCA mission: it specifies that a certain number of aircraft will leave a given airbase (and unit) at a specified time and fly to a given set of stations (described below). The refueling mission also specifies the total amount of fuel carried by the tanker, to be discoursed among the OCA missions which require refueling.

An orbit (ORB frame) specifies a location, during which refueling may be done. It may be seen as a "window" of refueling capability. In other words, the OCA missions which need refueling and the refuel-

ing missions which supply fuel use the orbits to constrain possibilities: it may not be possible to get refueling at any time during an orbit, but it is assumed impossible to refuel at a place or time not designated as an orbit.

Each refuel mission has several stations represented by the STN frames. The refueling mission must fly in sequence among its assigned stations for the duration specified by the start and end time of each station. Each station is a subinterval of an orbit during which the refueling aircraft is guaranteed to be in the orbit. In other words, to a refueling mission a station represents an obligation, to an OCA mission it represents an opportunity for refueling.

A service (SVC frame) then links up an OCA mission with a station and represents a single refueling event involving one OCA aircraft and one refueling tanker. Each individual service frame records the start time and duration of the service, as well as the fuel disbursement (the amount of fuel to be transferred) at that service.

1.5.2 KRS in FY84

The work and results in FY84 centered on two issues: (1) extension of KRS to more realistic planning scenarios, and (2) experimental implementations of more flexible planning strategies.

1.5.2.1 New Planning Scenarios

The automatic planning facilities carried over from KNOBS used a depth-first generate-and-test paradigm. This was adequate for the highly stereotyped version of an OCA mission then under investigation. In FY84 we began gathering knowledge and extending the KRS data base to encompass more realistic planning scenarios. As this effort progressed and as more types of missions and richer interrelations among missions -- both explicit mutual support and implicit competition for scarce resources -- were investigated, the inadequacy of any rigid problem-solving strategy became obvious.

As we continued to learn more about planning of realistically sized missions, as opposed to KNOBS's two and four-plane missions, the interrelationship between planning and replanning became clearer. Consider the following:

Planning refueling begins with SAC proposing strawman missions.
The tactical missions are planned against this
strawman proposal with variations negotiated as required.

A large mission involving perhaps hundreds of planes is planned
several different times to varying detail, with some details changing
while others are acted on. For example, on the day before the
mission flies, munitions and equipment have already been transported
to the anticipated airbases; that aspect of the plan cannot be
changed. At noon of the day the mission flies, some of the
aircraft may already be loaded and on the runway; the
ordnance cannot be changed, yet the enemy's activity
will certainly force some changes to targeting.

In these examples, we see both that "replanning" is an integral part of planning, and that the planning strategies used to formulate mission plans change over time and with the planning goals. That is, realistic planning involves both refinement and replanning. Our FY84 efforts to extend KRS led us to the belief that while the constraints on an OCA mission were constant over time, the strategy for meeting those constraints changed both with time and the purposes of the planner. Clearly, the strategy used for the

"strawman" refueling plans, which is based on the apportionment decisions, is different than the strategies used to negotiate changes for meeting the strike mission needs.

As an illustrative example, automatic planning of refueling was reimplemented about halfway through FY84. However, because of the rigid "autoplanning" facility, the OCA mission plan was completed and judged acceptable before autoplanning the refueling mission was begun. The result was frequently a "bad" OCA mission that went undetected because the constraint -- that the refueling mission and OCA mission must be at more or less the same time -- was never checked. Simply interchanging the order in which missions are planned leads to different but conceptually similar errors. Instead, what was required was a facility that would plan several OCA missions needing refueling up to but not including the timing of the mission, then plan the refueling missions around the target time windows, and only then complete the plans for the OCA missions.

Planning refueling brought to light several qualitative distinctions between KNOBS style missions and KRS style missions. Both missions are described by frames containing slots. However, in a number of dimensions KNOBS is qualitatively simpler than KRS as is shown in Table 1.1.

	KNOBS	KRS
SLOT FILLER	Atomic items from database	Atomic items and other mission frames that are "planned" by KRS
CONSTRAINTS	Among slots in single frame	Among slots in multiple frames, including constraints on frames as entities
INHERITANCE	Trivial mission AKO tree	Hierarchy of mission types
AUTOPLAN	Rigid	Planning use multiple interrelated frames
CONSTRAINT	Simple failure with minimal explanation	Failures may report VIOLATION suggestions for correcting difficulty

Table 1.1 Comparison KNOBS - KRS

We discovered the need for qualitatively different constraint violation reports because the major slot fillers could no longer be enumerated. Even KNOBS had difficulty enumerating things like "time" and "number of A/C"; special tricks were implemented. However, a more general solution is required when the slot fillers are themselves being planned by KRS.

In the later part of FY84, we found it was necessary to distinguish between "goals", which pertain to interaction of mission components (e.g., a "wild weasel" SAM suppression mission has the goal of keeping the SAM inoperative to support the strike force) and "metagoals", which pertain to the selection of planning strategies (e.g., if there is a metagoal of conserving fuel, then A/C should be flown from the closest airbase to the target; if there is a metagoal of maximizing penetration, then A/C should be flown from the farthest airbase from the target). We further found it necessary to distinguish between the plan structure -- which included mission frames and so-called "super-frames" like the package frame -- and

the "agenda" or "metaplan" structure that describes how to fill in the slots of the frames. Implementation of these ideas in KRS did not start until FY85; however, experimentation was done to verify our ideas.

1.5.2.2 Automatic Planning

In FY84, two alternative organizations for more flexible automatic planning were implemented in an experimental version for internal experimentation and evaluation: (1) "metaplanning" and (2) "agendas". Both organizations are based on the idea that expert planners know something about "how to plan" that goes beyond the constraints on the plan elements. To reiterate a previous example, there is no place on the templates for refueling frames to encode the fact that OCA mission planning should start with a "strawman" proposal of refueling flight timings. The "metaplanning" approach used a temporally oriented "script" that described how the plan would be executed. It used a process of "envisioning" to discover unanticipated failures, and a library of "metaplans" indexed by "metagoals" to construct corrections to the plans it was creating. The "agenda" approach exploited the basic similarity between three processes: enumeration of a fixed set of slot fillers, using a "generate-and-test" strategy on a sequence of slots, and generating alternatives based on why prior attempts were failing. Based primarily on the current set of plan goals but also taking into account the mission goals, an "agenda" would be selected that claimed to know how to plan the mission, fill the slot, etc. Naturally, different agendas would be selected for planning vs. replanning. The actual planning activity would be accomplished by sending messages to the agendas, which were implemented as a sophisticated use of the Zetalisp flavor mechanisms. The results of these two experiments will be discussed in later sections of this report.

1.5.3 KRS in FY85

There were three major changes introduced in FY85 as KRS evolved. First, the system was expanded to consider a larger planning element, the strike package. A package organizes primary strike missions, SAM suppression mission, and air escort missions to strike a single target. KRS plans each mission in the package and provides complete refueling support, including tanker mission scheduling. This required that KRS be capable of coordinating times between missions, and provided the first evidence of the increasing influence of interactions between missions in the planning activity. In recognition of these interactions, a coordinating entity, the ATO was constructed to hold all packages for a single day's targets.

The second change was a substantial increase in the number of targets planned, and therefore, an increase in the number of missions KRS needed to plan. Increasing the mission count led to new difficulties in allocating aircraft and the need to reason about aircraft alternatives and mission priorities.

The third change was the need for replanning in response to changing battle conditions, minimizing the impact on the overall plan. The KRS scenario assumes that the ATO is created on the day prior to plan execution. A battle change, such as an increase in SAM protection for a target on the day before the missions are to fly, requires an assessment of the impact of the change on the ATO and could lead to reallocation of aircraft. Consideration of these sorts of changes has led MITRE to focus on the resource allocation and reallocation problem for most of FY85. The goal was to develop those techniques for efficiently allocating resources during initial plan construction and during replanning. The competition for resources compounds the interactions between missions, and therefore, tightly couples all packages in the ATO.

1.5.3.1 Interrelated Mission Planning

It was recognized that solving the complex problems that arise from planning and replanning an air tasking order with limited resources requires a more elaborate reasoning mechanism than the one which was available in KRS at this time. The constraint based, bottom up view of planning knowledge in KNOBS which was sufficient for automatically detailing a single mission, does not contain the necessary information

for dealing with optimum resource allocation among several missions, reallocation of those resources when conditions change, or making decisions about mission priorities in a replanning scenario. All of these problems require a more global perspective on the set of interrelated missions.

The solution was to augment the KNOPS' single mission planning method in several ways: creation of strike packages that contain a strike mission together with support missions, construction of a resource tracking mechanism, and a metapanning component, so that KRS could reason, to some extent, about its own planning and replanning activity in solving the aircraft allocation problem and eliminating conflicts.

1.5.3.2 Demonstration System

An important question that arose in the KRS project concerned the architecture to be employed in the implementation of the FY85 demonstration system. The range of choices was narrowed by the decision to base the final system on the inherited KNOBS architecture used in KRS as implemented in FY84. The decision to keep as much of the existing KNOBS structure as possible was made for two main reasons:

- (1) KRS already displayed certain capacities which are desirable in a mission planning system, and the new system should not lose any of those capacities.
- (2) The AI techniques already developed, while inappropriate for extension to complex global mission planning and replanning, are efficient for detailing a partially constrained (by another agent) single mission.

Given the distributed nature of KRS's methods of storing information, handling constraint checking and choice generation, and inference, the problem became: how to add a metapanning capacity which is essentially a centralized and globally active process? The solution adopted was to use a simple version of a "distributed blackboard".

In the blackboard model, a community of experts cooperates on a task; the experts communicate through the use of a shared database called a "blackboard". The blackboard is often partitioned into areas which include a metapanning level, a planning level, a resource level, and so on. Each expert is responsible only for a certain portion of the task, and needs to access only certain parts of the blackboard, but can post information which is globally available to anyone who needs it. This sort of organization allows for both a partitioning of the task among various specialized experts and for a sharing of information among interacting experts whose areas of expertise either overlap or interface.

In MITRE's application of the blackboard model to KRS, the metaplanner surveys the world model, user goals, and air tasking order specifications, then determines any metagoals, subgoals, and air tasking order specifications, then determines any metagoals, subgoals, pieces of advice, restrictions, and relevant information which may expedite the planning process. All of this information is posted on the blackboard in appropriate places so that it is accessible to the lower level functions which perform various subtasks in KRS. These lower level planning functions were changed from their KNOBS form in order to enable them to make intelligent use of the goals and information posted by other parts of the planning system.

As an example, consider the ordering of possible candidates to fill the airbase slot of a mission frame. Suppose that at the ATO level, the resource goal of preserving fuel has been posted. The function which does the ordering of candidate airbases is alerted to this goal, and is intelligent enough to order the airbases by distance from the targets.

The blackboard also serves as a means of passing information to higher level planning activities. When some planning process encounters difficulties or conflicts, it can complain to the appropriate expert knowledgeable in fixing the particular type of problem in question. In their previous form, constraints had been fairly simple and not very useful for explaining failures or for automatically resolving those failures.

In FY85, constraints were augmented to return more information about their failures. Examples of such information include: alternate suggestions or strategies, the previous choices that are causing the conflict, and information which can help explain failures to the user.

A major part of the KRS project in FY85 involved the development of a formalism for expressing metaplanning knowledge about efficient resource allocation, and this formalism is described in section 4 of this report. The formalism was designed to satisfy two objectives: to provide a solution to the non time-critical replanning problem and to develop a general representation for strategies that will be useful in extending KRS to dynamic replanning. The formalism is:

- (1) able to express global strategies and heuristics, and to provide failure handling methods.
- (2) interpretable both procedurally and declaratively (to allow intelligent explanation by the system of its own activity).

The interpreter for this formalism was developed and is capable of maintaining a history of strategies and metaplans that are useful for plan generation and subsequent replanning. The potential of this formalism has been only partially explored in the tactical air domain because it was used only in the ATO and package planning levels of KRS, not in the planning of single missions (which continue to use the KNOBS inference mechanisms), nor in solving any dynamic replanning problems.

The end of the FY85 project has seen the construction of a demonstration system which exhibits prototype capabilities in the tactical air planning and replanning domain. This system extends the domain of concern to include a wider spectrum of problems. The KNOBS system had been limited to planning single OCA missions in isolation; the new system is capable of planning multiple packages (each package consisting of several interrelated missions including refueling support) and resolving resource allocation conflicts that arise when a whole ATO has already been planned.

The remainder of this report discusses the elements of the KRS project that represent a departure from the earlier KNOBS implementation. In section 2, issues associated with the man-machine interface are presented and this includes both lessons learned during the KRS implementation and the line of reasoning to explain why a specific approach was taken. Section 3 discusses the basic components of KRS and, in some cases, contrasts the KRS implementation to KNOBS. Section 4 reviews the relationship between planning, metaplanning, replanning, and strategies, and describes the implementation used in KRS.

SECTION 2 - THE KRS INTERFACE

2.0 INTRODUCTION

Conventional interfaces to computer systems have lacked the flexibility and convenience of natural communication. Learning and using a system with a natural interface enables a user to focus on the system's functionality instead of its protocols. The purpose of this project was to construct such an interface for the KNOBS mission planning program (and later the KRS follow-on project). Many subgoals were included in this purpose, among them, the construction of a robust parser and the development of a question answering facility that had a knowledge of user goals. We achieved some of these goals under this project; those that were not accomplished are still being pursued at MITRE under a follow-on project to 6070.

Our major accomplishment under this project was to construct a useful interface to KNOBS and KRS to demonstrate the potential of combining artificial intelligence techniques for language understanding with graphics as a means of expressing information. The most developed capability of this interface system is its ability to conduct an English conversation to assist in the planning of an Air Force mission. The interface answers a user's questions about targets and resources by constructing a data base query. It can currently express the answer by producing an English response, or by constructing a table, depending upon the type and amount of information to be presented. We also experimented with producing a map display to answer questions about geographical data. In addition to answering the user's question, the interface can interpret requests to plan a mission and update a display that describes a planned mission. A menu interface is also provided.

We introduced a number of facilities to make communication with a computer in English more practical. To ease the burden on typing, we included synonym and spelling correction capabilities, as well as some tolerance of poor grammar and punctuation so that a question such as "What AC do Hahn supply?" was interpreted as though it were typed "What aircraft does Hahn supply?"

Our interface produces a semi-canonical meaning representation of a sentence by combining the meanings of the words in a sentence. We created a formalism for expressing word definitions which was also used to describe the problem a mission planner is solving, as well as a stereotypical solution to his problem. Once the meaning of a question is represented, it is answered by rules that can construct a data base query from this meaning representation. We used a semi-canonical representation of meaning so that the question-answering and inference processes did not have to be concerned with the many ways of expressing the same idea in English.

2.1 HISTORY OF APE-II PARSER IMPLEMENTATION

APE-II is the largest and most complex of several natural language systems developed at MITRE specifically to serve as a front end for KNOBS. In 1980, an attempt was made to implement the interface by using an Augmented Transition Network (ATN) parser. An ATN parser makes an effort to "understand" a sentence in the way grammar is taught. It determines the part of speech for each input word--noun, adjective, etc.--and identifies noun phrases, the subject and object of the sentence, and so on. The network is a way of encoding English grammar. For an explanation of natural language parsing using an ATN, see [Wood70].

The ATN itself was only one part of the input processing. A morphological analyzer was used to identify individual words in the input, despite the presence of suffixes, spelling errors, or irregular inflections. It uses a dictionary to associate words with such information as parts of speech, tense, pluralization, and occurrence in an idiom. In KNOBS, the dictionary entry for a word was part of its property list. The frame data base acted as an extension of the dictionary--the name of any frame was taken to be a noun. If the frame had an An Individual Of (AIO) entry, it was a proper noun; otherwise, it was a common noun.

The syntactic analysis performed by the ATN resulted in the assignment of values to certain "registers" to record the essential information carried by the sentence for use by the semantic components. As an example, the sentence "What kind of control radar does BE50362 have?" results in the following register:

Register	Contents
SUBJ	(BE50362)
OBJ	((AIO AKO) ? CONTROL-RADAR))
VERB	HAVE
	QTYPE WHAT-TYPE

The subject, object, verb, and question type have been identified. The object is really a noun phrase: "a kind of control-radar." It is represented in roughly the form of a rule clause, with a question mark for the unknown frame.

The semantic components, for noun phrase resolution and question answering, include various strategies to answer queries or perform requested actions, such as printing rules or modifying the data base.

Because the ATN parser is a machine for recognizing a specified grammar for English, it could not handle ungrammatical utterances, or uncommon constructions that had not been incorporated into the grammar. It had particular difficulty with sentence fragments, which are common in conversation, such as "AND ITS SPEED?" after a previous query about an aircraft. The flexibility of a pattern-action production rule system was also missing.

2.1.1 ATN-Based Natural Language

An ATN is based on a transition network (more precisely, a finite-state machine) that "accepts" grammatically correct inputs. Expressing a grammar as an ATN consists of building a machine that recognizes acceptable inputs, as compared to a "transformational grammar" that describes acceptable inputs by giving a way to generate all and only those inputs. The difference between an ATN and a traditional finite-state machine is that the ATN can "jump to" or recursively invoke another finite-state machine on transitions from one state to another.

The statement that an ATN cannot accept "ungrammatical input" is somewhat misleading. In fact, for any list of acceptable inputs (which may or may not correspond to traditional English) there is a grammar. The difficulty is that the class of inputs that were supposed to be acceptable for KNOBS could not be described by a grammar (again, the grammar for English is just one of an infinite number of possible grammars) easily expressed using an ATN. It is meaningless to try to talk about a grammar without having some formal means of expressing that grammar. ATNs do not seem to be a good way to express the grammar for the inputs KNOBS needed to understand.

The KNOBS researchers gave up trying to express the desired KNOBS grammar using an ATN because of the enormous context-sensitivity required to tell if a random word or phrase was acceptable in the given discourse context. Whether the acceptability of such an utterance could or could not be expressed by an ATN is moot.

2.1.2 APE-II Development

Three major decisions made in the course of the KNOBS project determined the eventual shape of APE-II: (1) the decision to use natural language as a front end to the expert system, (2) the decision to use conceptual dependency theory as the basis for a natural language parser, and (3) the decision to integrate the parser with the rest of KNOBS. We shall discuss each of these decisions in detail.

Natural language has always been a major research topic in AI, and this is part of the reason for its choice as the medium for communicating with KNOBS. The developers of KNOBS and its sponsors welcomed the opportunity to explore this topic in the course of the KNOBS project. Much of the KNOBS effort went into its parser. MITRE is still engaged in natural language research in several contexts; the APE-II experiments have provided valuable information about both promising leads and pitfalls.

Other, more practical, arguments exist for natural language as the vehicle for communication with an expert system. Users typically like the IDEA of being able to rely on English; it is claimed that natural language is simpler to use than any other medium of communication with a machine. However, these claims are not always justified. Users communicating with KRS, for example, tend to rely on the menu driven command interface in order to get the system to carry out its task of planning missions. To issue commands, menus are actually simpler and faster than English. It is also hard to go wrong with a menu. No natural language interface currently in existence is sufficiently robust to prevent frequent and confusing misunderstandings between user and machine. After a few frustrating exchanges with a natural language parser, which seems stubbornly incapable of making sense of reasonable English sentences, a user might welcome the chance to turn to a nonverbal but foolproof means of communication such as a menu. Finally, users do not like to type; speech is natural, but typing is not. For these reasons, it may be true that sophisticated menu driven interfaces are more suited to user needs than is language.

However, there are still places where language is desirable. User questions, for example, cannot always be anticipated, nor can a system predict when a user might need to ask a certain question. As a result, it is hard to deal with queries and responses through a static medium such as a menu. Also, a good natural language system enables a user to exploit the use of pronouns and ellipsis, two common and powerful linguistic devices without which verbal communication becomes clumsy, tedious, and unnatural.

Another reason to support natural language interfaces to expert systems is sometimes overlooked. In the future, speech understanding will become a reality, and when it does, speech will become the preferred means of communication with all sorts of automated systems. Parsers must be ready to understand what is said. For this reason, and for the others mentioned above, one can say that the initial decision to use language as the main vehicle for communication with KNOBS was justified.

Furthermore, this decision has paid off. APE-II has not solved all the problems associated with natural language interfaces, but it did solve several, and even its failures have yielded valuable insights about the weaknesses of certain knowledge representation schemes and parsing strategies. APE-II served as the basis of computational linguistics at MITRE for years and has been the bridge to a new set of projects that will use the lessons learned from APE-II in a new approach to natural language research.

Once the decision to use natural language had been made, the KNOBS team had to decide on a parsing strategy. After a brief period of experimentation with makeshift pattern matchers and ATNs, they settled on conceptual dependency (CD) theory to provide the theoretical foundation for the KNOBS parser. We now believe that this choice was a mistake, as we will show below. However, hindsight provides a clarity of vision unavailable to those who must solve problems as they appear, and there were excellent reasons for the KNOBS development team to choose a CD parser at the time they did.

The mid-1970s was a bad time for theoretical linguistics; the standard theory of Noam Chomsky had begun to lose ground to its various critics, but as yet no synthesis of competing ideas had emerged to

permit another theory to take its place. Furthermore, there was great confusion over the role semantics played with respect to syntax. The field was generally split into two camps: the theoretical syntacticians, who believed in rigorous analysis of syntactic structure, usually based on some set of grammatical rewrite rules, and a group which could be called the anti-syntacticians, represented in large part by Schank, Abelson, and their followers.

The anti-syntacticians believed that linguistic comprehension proceeds in a bottom up process as speakers and hearers derive word meanings in the context of a discourse. Structure is relatively unimportant in this model, but the representation of meaning is crucial. Schank and Abelson developed a theory of such representation, conceptual dependency theory, which had two basic tenets: all words can be decomposed into a small set of semantic primitives and the meaning of utterances can be represented as networks of CONCEPTS linked together in a CONCEPTUAL DEPENDENCY network. The concepts themselves are semantic primitives with associated arguments such as ACTOR, OBJECT, GOAL, and INSTRUMENT. Nouns are called PICTURE PRODUCERS and are usually treated as nondecomposable units.

Schank and his followers implemented this theory in several text understanding systems during the mid and late 1970s. One should take note of the fact that their theory is actually alinguistic; that is, it embodies no model of language. Rather it is a model of semantic representation. The conceptual dependency school produced a theory that skirted the theoretical controversies then raging in the field of linguistics and also had an implementation to back it up. This gave their theory a weight that most competing theories lacked at the time. Their parsers, SAM, MARGIE, and PAM, were slow and limited in scope, but at least they appeared to be making significant progress.

Theoretical linguistics pulled itself together in the late 1970s and early 1980s, and more and more AI researchers built good parsers that had at least some structural component based on linguistic theory. Also, CD based parsers began to show their limitations more clearly as they were extended over the years. By the time all this became clear, MITRE was locked into APE-II and isolated from recent advances both in linguistics and in semantic research. We now believe that APE-II should be judged as a product of 1970s technology whose limitations are largely due to weaknesses in its theoretical foundation.

One other decision about the KNOBS front end must be mentioned: the decision to integrate the parser with the expert system itself. A thorough discussion of this is found in "KNOBS The Final Report (1982)" [KNOBS82] by Richard Brown, whose comments about this issue represent a consensus of the MITRE natural language group. In some ways, it made life easier for the developers to rely on the intertwining of APE-II and KNOBS, but the decision as a whole was a bad one. Modularity is a fundamental principle of good software design; without it, portability, maintainability, and extensibility all suffer. In the case of APE-II, this truism has been born out. Not only is the parser completely nonportable and hard to maintain by anyone who does not know the entire APE/KNOBS system, but it also has suffered performance decay as a result of its overly close relationship with KNOBS. The APE-II developers have frequently observed that changes to KNOBS, especially in the area of constraints, have destroyed functionality in APE-II. Tracking down the reasons for such problems and fixing them have been a developer's nightmare. Once again, the MITRE AI researchers have learned from a mistake, and current linguistic interface work at MITRE is being guided by concern for modularity.

We now put aside historical considerations and examine APE-II from the point of view of implementation. This examination is followed by an evaluation of APE-II's performance as an interface that discusses the reasons behind its successes and failures. Included in this section is a detailed critique of CD theory from an implementation point of view.

2.2 IMPLEMENTATION OF APE-II

APE-II is a classic implementation of a CD parser along the lines of SAM. It represents state of the art natural language technology of about 1975. A semantically driven parser, APE-II relies on a dictionary of

word meanings to guide the parse. It lacks any independent syntactic component; any information about syntax is buried in dictionary entries for word meanings. APE-II has a simple morphological component to strip off affixes, but it does not attempt any derivation. APE-II makes no attempt to capture the relationship between "destroy" and "destruction", for example.

APE-II evolved from A Parsing Experiment (APE), a parser used by the Distributable Script Applying Mechanism (DSAM) and Academic Counseling Expert (ACE) project at the University of Connecticut [Cullingford82]. APE is based on the CD parser [Birnbaum81] with the addition of a word sense disambiguation algorithm.

In CD, word definitions are represented as requests, a type of test-action pair. The test part of a request can check lexical and semantic structures or connect CD structures, and activate or deactivate other requests. The method available to select the appropriate meaning of a word in CD is to use the test part of separate requests to examine the meanings of other words and to build a meaning representation as a function of this local context. For example, if the object of "serve" is a food, the meaning is "bring to"; if the object is a ball, the meaning is "hit toward." This method works well for selecting a sense of a word which has expectations. However, some words have no expectations and the intended sense is the one that is expected. For example, the proper sense of "ball" in "John kicked the ball." and "John attended the ball." is the sense that the central action expects.

The word definitions of APE-II are also represented as requests. A concept called a VEL is used to represent the set of possible meanings of a word. When searching for a concept with certain semantic features, an expectation can select one or more senses from a VEL and discard those that are not appropriate. In addition, APE-II can use expectations from a contextual knowledge source such as a script applier to select a word sense. Each script is augmented with parser-executable expectations called named requests. For example, at a certain point in understanding a restaurant story, leaving a tip for the waiter is expected. The parser is then given a named request which could help disambiguate the words "leave" and "tip," should they appear.

2.2.1 APE-II

A word definition in APE-II consists of the set of all of its senses. Each sense contains a concept, i.e., a partial CD structure that expresses the meaning of this sense, and a set of conceptual and lexical expectations.

A conceptual expectation instructs the parser to look for a concept in a certain relative position which meets a selectional restriction. The expectation also contains a selectional preference, a more specific, preferred category for the expected concept [Wilks72]. If such a concept is found, the expectation contains information on how it can be combined with the concept that initiated the expectation. A lexical expectation instructs the parser to look for a certain word and add a new, favored sense to it. This process is useful for predicting the function of a preposition [Reisbeck76]. The definition of a pronoun utilizes a context and focus mechanism to find the set of possible referents which agree with it in number and gender. THE PRONOUN IS THEN TREATED LIKE A WORD WITH MULTIPLE SENSES. The definitions of the words "fly," "eat," and "A/C" are shown in figure 2-1.

```

(DEF-WORD A/C (SENSE (AIRCRAFT))
  (SENSE (AIR-CONDITIONER)))(DEF-WORD EAT (SENSE [EAT ACTOR (NIL)
  OBJECT (NIL)
  TO (*INSIDE* PLACE (*STOMACH* PART (NIL)
  EXPECTATIONS ([IF (IN-ACT-SPOT #ANIMATE)
    THEN ((SLOTS (TO PLACE PART)
      (ACTOR)
      [IF (IN-OBJ-SPOT *PP*)
        PREFER (#FOOD)
        THEN ((SLOTS (OBJECT)))(DEF-WORD FLY (SENSE (FLY
OBJECT (NIL)

  ACTOR (NIL)
  INSTRUMENT ($FLY)
  TO (*PROX* PLACE (NIL))
  FROM (*PROX* PLACE (NIL)))
  EXPECTATIONS ([IF (IN-ACT-SPOT AIRCRAFT)
    THEN ((SLOTS (OBJECT)))
    ELSE (IF (IN-ACT-SPOT BIRD)
      THEN ((SLOTS (ACTOR) (OBJECT)))
    LEXICAL-EXPECTATIONS ((TO (MAKE-DEF (OB-PREP *PP*)
      (TO PLACE)
      (*LOC*)))
      (FROM (MAKE-DEF (OB-PREP *PP*)
        (FROM PLACE)
        (*LOC*))))))

```

Figure 2-1. APE-II Dictionary Definitions

The definition of "A/C" states that it means AIRCRAFT or AIR-CONDITIONER. APE-II uses selectional restrictions to choose the proper sense of "A/C" in the question "What A/C can fly from Hahn?" On the other hand, in the sentence "Send 4A/C to BE70701." APE-II utilizes the facts that the OCA script is active and that sending aircraft to a target is a scene of that script to determine that "A/C" means AIRCRAFT. In the question "What is an A/C?" APE-II uses a weaker argument to resolve the potential ambiguity. It utilizes the fact that AIRCRAFT is an object that can perform a role in the OCA script, while an AIR-CONDITIONER cannot.

The definition of "fly" states that it means a kind of physical transfer. The expectations associated with FLY state that the actor of the sentence (i.e., a concept which precedes the action in a declarative sentence, follows "by" in a passive sentence, or appears in various places in questions, etc.) is expected to be an AIRCRAFT in which case it is the OBJECT of FLY or is expected to be a BIRD in which case it is both the ACTOR and the OBJECT of the physical transfer. This is the expectation which can select the intended sense of "A/C." If the word "to" appears, it might serve the function of indicating the filler of the TO case of FLY. The word "from" is given a similar definition, which would fill the FROM case with the object of the preposition, which should be a PICTURE-PRODUCER but is preferred to be a LOCATION.

The definition of "eat" contains an expectation with a selectional preference that indicates that the object is preferred to be food. This preference serves another purpose also. The object will be converted to a food if possible. For example, if the object were "chicken" then this conversion would assert that it is a dead and cooked chicken.

We will first discuss the parsing process as if sentences could be parsed in isolation and then explain how it is augmented to account for context. The simplified parsing process consists of adding the senses of each word to an active memory, considering the expectations, and removing concepts (senses) not connected to other concepts.

Word sense disambiguation and the resolution of pronominal references are achieved by several mechanisms. Selectional restrictions can be helpful to resolve ambiguities. For example, many actions require an animate actor. If there are several choices for the actor, the inanimate ones will be weeded out. Conversely, if there are several choices for the main action, and the actor has been established as animate, then those actions that require an inanimate actor will be discarded. Selectional preferences are used in addition to selectional restrictions. For example, if "eat" has an object which is a pronoun whose possible referents are a food and a coin, the food will be preferred and the coin discarded as a possible referent.

A conflict resolution mechanism is invoked if more than one concept satisfies the restrictions and preferences. This consists of using "conceptual constraints" to determine if the CD structure that would be built is plausible. These constraints are predicates associated with CD primitives. For example, the locational specifier *INSIDE* has a constraint which states that the contents must be smaller than the container.

The disambiguation process can make use of knowledge structures that represent stereotypical domain information. The conflict resolution algorithm also determines if the CD structure that would be built refers to a scene in an active script and prefers to build this type of conceptualization. At the end of the parse, if there is an ambiguous nominal, the possibilities are matched against the roles of the active scripts. Nominals that can be a script role are preferred.

2.2.2 APE-II Dictionary

The heart of APE-II is its dictionary, which, with the KRS list of frames, defines all the words that APE-II understands. The frames represent the objects of interest within this world; they are the nouns that APE understands. The dictionary defines all the non-nouns. In the case of APE-II, the latter are almost all verbs with the addition of three adjectives, the articles and conjunctions, and two adverbs. APE-II's dictionary is small, about 120 words total, but the entries are typically quite complex, with many lexical and structural expectations associated with each word sense. Because of the crucial importance of getting these expectations "exactly right," adding a new word to APE-II's dictionary can be very difficult. There is no canonical way of setting up these expectations; one usually derives them by observing the behavior of the parser as it encounters the word in question in different contexts. Trial and error play a large role.

Despite this, it is not too hard to define new verbs for APE-II. Nor, is it hard to add new nouns since these are, for the most part, handled separately in the FRL. However, there is a reason so few adjectives and adverbs appear in the APE-II dictionary: they are all but impossible to define adequately in terms of CDs. CD theory grew out of case grammar, a semantic theory based on the belief that it is possible to understand sentences by reducing them to actions and case fillers for those actions such as agent, instrument, location, and object. CD is thus *VERB ORIENTED*. A look at the semantic primitives proposed by Schank shows that they fall into this pattern; the primitives are primitive states and actions, that is, primitive verbal elements. CD was built to handle verbs, and it works fairly well at reducing them to primitives and selectional restrictions, but it has little to say about other types of words. This problem is serious for the dictionary designer who must decide how to represent something like the adverb "recently" in a way that is consistent with the rest of a CD parser. In APE-II, the developers never did solve this problem, and it remains extremely difficult to define anything but verbs.

For this reason, a proposed addition to APE-II's functionality, an interactive word definition package, never worked satisfactorily. The idea was to enable a user to define a new word in the course of a session with KRS. However, the process of specifying the definition was so tricky it made the plan unworkable.

Much of the effort and thought that went into making APE-II work went into the verb definitions. The lexical and structural expectations and selectional restrictions used to disambiguate word senses contain a great deal of domain-specific information about word usage in KRS. This knowledge, however, is procedurally, rather than declaratively, expressed, leading to all the problems associated with similar non-declarative designs. One problem is massive redundancy; many word senses rely on structural expectations that arise from the same fact about something in the KRS domain. Another problem is that there is often no way of obtaining needed information about the domain outside the context of disambiguating a word's meaning. Finally, maintenance of the dictionary becomes expensive because changes in the domain must be reflected in the expectations associated with dictionary items, and there is no alternative to tracking down each relevant expectation "by hand" every time such a change must be made.

Another vexing problem related to maintaining the dictionary in APE-II arises from the lack of canonicity in definitions. As shown below, the reliance on structural expectations to guide a parse dooms an implementor to depending on a set of ad hoc procedures. There is no guarantee that any two developers will define the same word the same way; in fact, it would be surprising if the two wrote identical procedures to serve as structural expectations. As a result, it is often confusing to look back and try to change a dictionary entry since the developer who created it may be long gone, and the procedural nature of the definition may make it impossible to figure out the reasoning behind it. Semantics is not an exact science, and its major issues remain topics of debate; anything that makes it harder to deal with these issues should be avoided.

2.2.3 SCRIPTS and User Goals

APE-II was implemented as a bottom-up parser, which used its dictionary to build up concepts from the meaning of the words in an input structure. APE-II could understand many user commands and queries in the sense that it could construct a CD to correspond to them and then match that CD to a KRS response. However, APE-II could not "understand" a question's point; it lacked any mechanism for deriving a user's motives for asking a question. This meant that, in many cases, APE-II could not evoke an efficient, cogent response to such a question. An example that shows how much of a handicap this lack of in-depth understanding can be is provided by the following (the user is planning a strike mission with KRS):

User: What planes does Hahn have? KNOBS: F-111 s, F-4Gs, and C-130s.

Of these planes, C-130s are unsuitable as a choice for an attack plane. If APE-II could have inferred that the user was asking a question so as to plan a strike mission, it would not have listed the C-130s in the response but would have confined the list to appropriate attack planes.

We decided to implement SCRIPTS, a context representation scheme developed by Schank and his students, to solve the problem of inferring user goals. SCRIPTS represents, in declarative form, stereotyped knowledge about the objects and actions in a given domain. SCRIPTS is divided into scenes, small chunks of description about specific actions in the realm of discourse. Scenes are connected to each other by temporal links and by enabling and disabling links. In a script that describes the execution of a strike mission, for example, one might have a scene that describes the attack plane's takeoff, a scene that described its flight to the target, and a scene that described its attack upon the target. Each of these scenes would be temporally linked, and the earlier scenes would enable the later.

We believed that by figuring out what scene you were in, you could figure out user goals in asking a question. We developed a design strategy based on this belief and drew up a small script that described

the execution of an air attack mission. However, the full script was never implemented, and APE-II never developed the ability to determine user goals. We still believe that scripts have an important role to play in determining user goals and are committed to a script-based implementation of a parser other than APE-II. Because the implementation of scripts never progressed beyond embryonic form in APE-II, one cannot draw any conclusions about SCRIPTS' utility from the APE-II experience.

Consensus of the research staff, however, is that we had made at least one error in the initial approach to SCRIPTS. The script we designed mirrored the EXECUTION of a mission. The goal of the user is to PLAN a mission. Had we designed a planning script rather than an execution script, we might have found that determination of user goals was as simple as identifying the current scene of the script. That is, if the script were in the "plan the choice of attack aircraft scene" and the user asked a question about aircraft, one could assume from the scene that his goal was to choose an attack plane.

2.3 EVALUATION OF APE-II PERFORMANCE

APE-II should be viewed as an experiment in natural language research. It is by no means the final word on natural language parsing; some of its results offer promising leads for further work while others must be regarded as dead ends. It is our belief that the APE-II experiment provided a rigorous test of CD theory and that the most serious APE-II failures resulted from weaknesses inherent in that theory. As a result, we believe that any evaluation of APE-II performance must be accompanied by a thorough discussion of CD theory with respect to its implementation in a parser.

APE-II is fairly slow, although not unacceptably so. The user might have to wait several seconds for a response to a valid input, but this is no worse than the wait he must experience while a mission window is being created. The bottom up resolution of word meaning that APE-II uses to parse is an inherently slow algorithm since it requires creation of numerous alternatives and choice among them; in this respect APE-II is as fast as any CD parser and faster than some, such as SAM.

APE-II has two great strengths: it handles lexical ambiguity well, and it uses a powerful spelling corrector which greatly eases the burden on the user who must rely on typed input as his only means of communication with KRS. The importance of resolving lexical ambiguity cannot be overstated since it pervades natural language. APE-II rarely chooses the wrong sense of a word; when it sees the phrase "How many planes can fly to the target?" it does not interpret the word "fly" as a kind of insect, nor does it interpret "planes" as a geometric abstraction. APE-II demonstrates that the use of selectional restrictions to guide the parser's choice of word sense works, and works well. Natural language work at MITRE will make use of this experimental result.

However, APE-II cannot handle several important linguistic phenomena. These include pronominal reference and anaphora, ellipsis, and identification of the referent of a noun phrase. The designers of APE-II attempted to deal with each of these phenomena, but they never implemented a general solution to the problems they pose. The following paragraphs discuss each of these issues in turn.

APE-II's worst failing is its general inability to find the referent of a noun phrase. Consider the following fragment of user/expert system dialogue:

User: Refuel three F-4Cs at Hahn. KRS: Planes refueled.
User: Send three aircraft to Dresden. KRS: What type of aircraft would you like to send?
User: The three planes I refueled at Hahn.

There is no way for APE-II to find the objects to which the phrase "The three planes I refueled at Hahn" refers. This is a serious problem since people freely refer to objects by noun phrases which

describe their properties and history. APE-II, however, has no way to record and update the events which an object in the KRS world might have undergone. The problem, therefore, goes beyond matching an input description to an object in the data base; it involves recording changes to such an object as they happen.

This failing is part of a more general problem in APE-II, one that also involves pronouns. APE-II, in general, cannot find pronominal antecedents in a discourse. One very limited case in which APE-II does match a pronoun to an antecedent is illustrated by the following dialogue:

User: What kinds of planes does Hahn have? KRS: F-4Cs, F-4Gs, and C-130s.

User: Which of its fighters can reach the target? KRS: F-4Gs can reach the target from Hahn.

In this one case, APE-II matches the pronoun "its" to the location "Hahn". However, the solution to the problem of pronominal reference used to understand the fragment above does not generalize to any other case. This is because the solution involves two crucial limitations: (1) APE-II can only refer to the IMMEDIATELY PRECEDING USER INPUT, and (2) APE-II treats the word "its" as a possessive locative; that is to say, the definition of the word triggers the application of a rule that checks for an unfilled location slot on the word "its" modifies. If the slot is empty, and if there is a location (read airbase) in the IMMEDIATELY PRECEDING input, APE-II will fill the slot with that location. One can see at once that this solution will not generalize to any other case and will not work for the vast bulk of anaphoric problems. Indeed, it has at least one weird side effect as the following fragment of dialogue shows:

User: What kinds of planes does Hahn have? KRS: F-4Cs, F-4Gs, and C-130s.

User: Which of its fighters can reach the target? KRS: F-4Gs can reach the target from Hahn.

User: What ordnances do its fighters carry? KRS: Unable to parse. Could you please rephrase that?

The designers' habit of applying non-extensible solutions to major linguistic problems eventually undercut APE-II's utility as an interface. Without a model to describe anaphora and reference, APE-II's designers were unable to derive a general solution to these problems. In fact, the only theoretically complete treatment of any linguistic problem in APE-II was its treatment of word meaning; syntax, pronominalization, reference, and ellipsis were all tackled piecemeal. The result was a system that sometimes worked and sometimes didn't, and a user could never predict whether his input would be understood or not.

We believe that this piecemeal approach was at least partially a reflection of the anti-linguistic bias inherent in CD theory. CD theory is alinguistic in the sense that it incorporates no explicit model of language. Rather, CD theory is a theory of knowledge representation, and, in part, a psychological theory. When it deals with knowledge representation issues, such as word meaning, CD theory works fairly well, but when it attempts to handle specifically linguistic phenomena such as ellipsis and pronominalization, it fails. Furthermore, for a variety of reasons which we will soon discuss, CD theory is IMPLEMENTATIONALLY DANGEROUS. That is, it lends itself to implementations in which key issues are neglected and key mechanisms mishandled.

2.3.1 CD Theory is Implementationally Dangerous

The original scheme of CDs discussed by [Schank & Colby73] portrayed the CD as existing in a multi-dimensional semantic network which incorporated the hierarchical links representing decomposi-

tion into primitives in addition to other, non-hierarchical dependency links between concepts. The diagram in figure 2-2, taken from Schank and Colby's *Computer Models of Thought and Language*, illustrates the network structure of the CD for the decomposition of one sense of the word "threaten." In English, this diagram would mean: "X communicates something to Y with the consequence that Y now believes that Y doing something will have the consequence of X doing something else that has the consequence of Y being hurt (actually, Y's health will decrease)."

Early implementations of CD-based parsers demonstrated the power of this approach to semantics, but they also provided warnings about its dangers. A good example is MARGIE, developed by Schank, Riesbeck, Rieger, and Goldman. MARGIE relied heavily on semantic and structural expectations associated with words. A close look at these expectations shows that they are not that different from semantic actions used in more structural approaches to parsing and thus are not so much declarative as procedural. For example, Schank writes of word expectations in MARGIE, "Associated with an expectation is a set of actions that are performed if an expectation is fulfilled. In general these actions can be any kind of behavior available to the entity doing the comprehending."

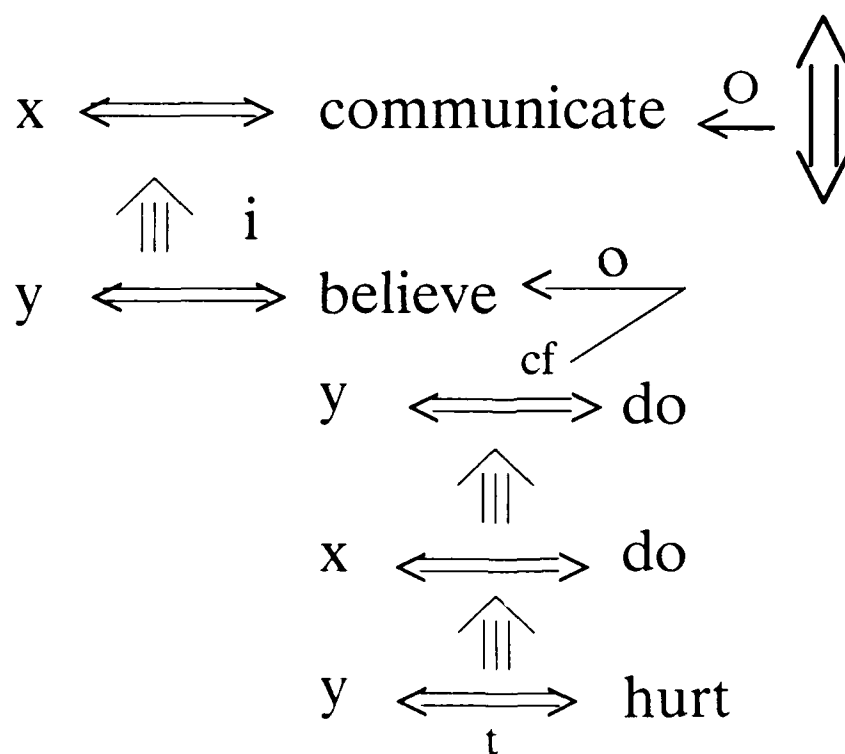


Figure 2-2 CD Representation of "X Threatens Y"

"A SAM threatens the OCA"

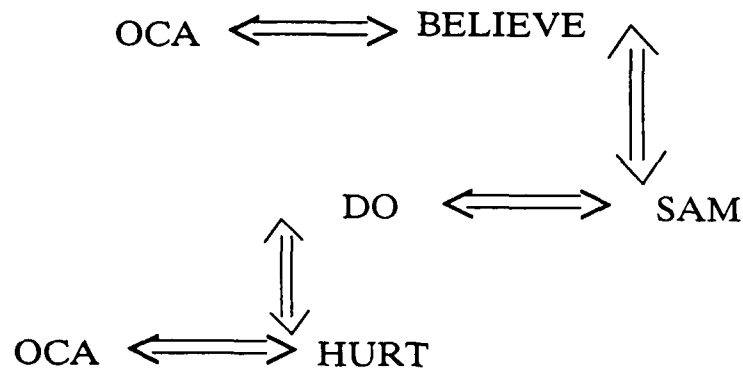


Figure 2-3: Network Representation

```

(BELIEVE
  (OBJECT
    (ATTACK
      OBJECT (OCA)
      AGENT (SAM)
      ACTOR (OCA)))
  )
  
```

Figure 2-4: Tree Representation

MARGIE also depended on an inference mechanism that generated every possible inference it could about an input sentence. This mechanism was powerful, but the lack of any ability to restrict the number of inferences produced led to fatal implementation difficulties. Later implementations of CD-based systems overcame the second difficulty by relying on context representation schemes such as scripts. This was the approach taken by the developers of KNOBS and its natural language front end, APE. However, the first problem, the dangers of relying on procedural actions associated with structural expectations, not only was not resolved, it was not even discussed as a problem by many researchers.

APE-II uses a dictionary of word sense meanings to build an internal conceptual dependency for an input sentence. As in MARGIE, expectations associated with each word sense are used to narrow possible interpretations. The output of the parser (which cannot now parse our example sentence) was a conceptual dependency represented as a tree.

We believe APE's major weaknesses are due to these factors:

CDs were represented as trees (i.e., implemented as LISP lists whose first element is the name of the concept and whose other elements are attributes). This doomed APE's ability to match question patterns against the question's CDs for reasons we discuss below.

Word sense disambiguation depended on structural expectations associated with each vocabulary item. As we shall show, this limited the parser's ability to deal with ungrammatical or fragmented input.

Each word, and each sense of each word, is unrelated to anything else. For example, "threaten" in the CD diagram and "threaten" in our sample sentences are clearly different but related senses of the lexeme. In APE-II, there is virtually no relation between the two.

The set of "primitive" relational and action concepts was large. This choice was inevitable, and also destroyed APE's flexibility. We'll present a "doomed if you do; doomed if you don't" analysis showing why this dilemma cannot be resolved in the classic CD approach.

2.3.2 Do Not Represent Conceptual Dependencies as Trees

The fact that dependencies among concepts were not restricted to compositional links is crucial, but this is something that has been overlooked by a number of researchers who have attempted to implement parsers based on CD. The source of the overwhelming temptation to represent CDs as trees is clear: they're easier to build that way, and a recursive response mechanism virtually demands a tree-like organization.

Trees are purely hierarchical structures with links from parent node to child subnode running in one direction. When CDs are represented as trees the only links among concepts are the direct links from CD to constituent; reverse links and other interrelationships are not

represented. The distinction between "tree" and "network" is technical but crucial--the contrasting representations of two CDs shown in figures 2-3 and 2-4 makes this distinction clear.

This reliance on trees is, in fact, one of the worst weaknesses of APE's implementation. Rather than maintaining the net structure that relates concepts in many different ways, APE-II implemented CDs as tree structures whose roots were semantic primitives. Lateral links were simply not used, with the result that all relationships between concepts were reduced to decomposition. In APE-II, only one level of branching was implemented so that the trees were basically flat structures.

Other CD parsers also rely on flat tree structures (LISP lists) to represent word definitions. For example, figure 2-5 contains a definition of the word "crash" taken from a CD parser discussed as a model for researchers by Birnbaum and Selfridge in *Inside Computer Understanding*. This approach failed to exploit the true power of CDs to express relationships between word meanings. Because the only option available was a tree-oriented definition in terms of a root and a leaf, developers inevitably succumbed to the temptation to write recursive descent matchers against the definitions. This meant that concepts with considerable meaning overlap but which did not share the same tree structure could not match one another unless the matching algorithm moved from simple recursive descent to more complex pattern matching. As a result of this weakness, developers either had to warp their recursive tree matchers or overlook meaning generalizations.

Definition of "Crash"

```
(DEF CRASH
  (REQUEST REQ14
    [TEST: T]
    [ACTIONS:
      "SET STRO TO (PROPEL ACTOR (NIL)
        OBJECT (PP CLASS (PHTSOBJ) TYPE GROUND))
        PLACE (NIL))
      ADD STRO TO C-LIST
      "ACTIVATE
      (REQUEST REQ15
        [TEST: "FIND A PP WHICH IS A PHYSICAL OBJECT
          PRECEDING STRO ON THE C-LIST"
        [ACTIONS: "PUT IT IN THE ACTOR SLOT OF STRO"])]
      (REQUEST REZ16
        [TEST: "FIND A PP WHICH IS A LOCATION ON THE C-LIST"]
        [ACTIONS: "PUT IT IN THE PLACE SLOT OF STRO"])]])
```

Figure 2-5

Tree matchers know where to look for the start of their target pattern: the root. Network matchers do not know where to look; they must find the pattern by searching throughout the network. This is why they are computationally costly. However, the more one warps the kind of matcher used in APE-II the closer one moves toward a network matcher rather than toward a tree matcher. On the other hand, if one does not move in this direction, one has to capture meaning relationships by putting some kind of annotation

into the vocabulary entries. A more effective approach to the use of CDs is to follow the original CD scheme as proposed by Schank, in other words to embed the concepts in a network from the start. The network becomes the declarative component of the knowledge base, as it is in Schank's original model.

One can also write network patterns to be matched against the network resulting from a parse. In APE-II, even if the question "Does the SAM threaten the mission" could be answered, there would be no chance of answering the very similar "Can the SAMs hurt the mission," even though the central part of the network is (or, at least, ought to be) identical in both questions. If built as trees, the two questions have completely different structures.

2.3.3 Reliance on Structural Expectations Carries a Heavy Price

Objections to relying on structural expectations are based on more than aesthetics. One of the advantages claimed by CD advocates is that CD-based parsing allows the system to understand ungrammatical or fragmented sentences. Because the parsing is driven by word meanings rather than by syntax, ungrammatical but semantically meaningful input should be understandable. However, as the need to use lexical and structural expectations grows, the power of a CD-based parser to understand ungrammatical input declines. Developers are caught in a cleft stick; if they forego the use of structural expectations, they lose their most reliable guide to lexical disambiguation. If, however, they rely heavily on structural expectations they find themselves restricted to those sentence structures from which such expectations were derived. Thus, one of the key "promised benefits" gained by using a CD approach is unavailable to most CD developers.

In text understanding systems that deal with complete and usually grammatical sentences, this problem might be overlooked. APE-II, however, serves as the front end to an expert system; users engage in a dialogue with that system, and they frequently resort to sentence fragments. The developers of APE-II, in fact, had that in mind when they chose to implement a CD-based scheme. Their hopes of exploiting that scheme's ability to handle ungrammatical input could not be realized because of their dependence on structural guides to disambiguation.

2.3.4 Do Not Ignore Syntactic Relationships

APE-II had a very primitive facility for defining synonyms: a lexical substitution took place after the string was read from the terminal, but before the scripts or parser saw the characters. This meant that word similarity was an all or nothing thing: either two words were unrelated or one meant exactly the same as the other.

The inability to define word and word-sense similarities would not have been debilitating, except that adding new vocabulary was not only difficult, but actually became more difficult as the vocabulary increased. Since none of the burden of parsing could be placed on a syntactic component (the implementors dogmatically insisted that "there is no need for syntax"), all interword relations had to be encoded in so-called lexical and structural expectations. As the vocabulary increased, so too did the complexity of these expectations.

As proof of the syntactic nature of word expectations, consider the admission of Birnbaum and Selfridge in *Inside Computer Understanding* that "In syntactic analyzers, the expectations are derived from a grammar. In conceptual analyzers, the expectations are governed instead by the incomplete structures representing the meaning of the input." In the same work, the authors give an example of a word definition (fig. 2-5) in which a host of syntactic and semantic information is buried inside a set of expectations, tests, and procedures. Not only does this approach toward word definition limit the parser's ability to understand ungrammatical input, it also makes it exceedingly difficult to figure out how to add a new word to the system. A look at the definition for "crash" in figure 2-5 is not reassuring in this respect.

In APE-II, the difficulty of defining words became a serious obstacle to anyone trying to extend the parser to a new domain since APE-II relied on a dictionary of baroque complexity. Moreover, the practical experience of dealing with vocabulary in APE-II showed that defining a new word entailed entering a debugging loop in which the developer tried out sentence after sentence using the newly defined word until he finally tweaked the definition into acceptability.

The inattention to nonsemantic word relationships also accounts for APE's inability to explain why it failed to parse a sentence: it didn't pay attention to that most basic of syntactic categories: part of speech. Nor did APE-II have any general way to identify relational categories such as "subject" or "main verb." (Actually, it tried; hidden in the parser's primitives were lines of code that set internal flags that were examined by the lexical and structural expectations. These flags constituted an ill-conceived ad hoc syntactic component). The result, however, is that there is simply no way for it to say "Gee, that sentence has no verb."

It happens that in the limited context of OCA missions, the verbs "hit," "bomb," "strike," and "take out" are all synonyms. Couple this fact with a good morphological decomposition component, and a system that uses both syntactic and CD-based semantic parsing can realize a quantum leap in word use flexibility. But you must have syntax!

2.3.5 Believe Not in Primitives: Doomed If You Do

APE-II wisely did not use an "official" set of 11 primitive concepts (i.e., ATRANS, PTRANS, MTRANS, and we can argue about the rest). From the viewpoint of APE-II, KNOBS, and its users, an aircraft taking off from a runway is a "primitive" action in the sense that it is never expanded (in this application). There are two excellent reasons for not expanding concepts into their defining semantic networks: first, some subtle meaning is always lost in such a transformation (consider "John kicked the ball" versus "John propelled the ball with his foot"; something is added, something is lost, and the two are never completely equivalent); and secondly, there is a high cost in computational complexity if one does.

If every concept is expanded into its "equivalent" semantic network, then the cost of matching pattern against parsed "meaning representation" increases. If the CD is represented as a tree, then the computational complexity of the match grows linearly in the size of the parse. That's not too bad.

We have already shown that to capture meaning generalizations properly, one must rely on networks rather than on trees. However, if the CD is represented as a true network (which must be done) then the complexity of matching grows as a high degree polynomial of the size of the parse and exponentially in the size of the largest pattern. That kills performance.

If CDs are represented only in terms of primitive concepts, the natural language interface is either doomed by its computational complexity or doomed by the weaknesses inherent in using trees to represent meaning.

2.3.6 Believe in Primitives: Doomed If You Don't

Suppose I tell you that "The SAM at MERSEBURG-B threatens OCA1001." Now I ask "Can the SAM at MERSEBURG-B attack OCA1001?" Pretend that APE-II can parse both sentences. Unless it expands CDs to a small set of primitives, it will not generally be able to match the question to the assertion because the semantic relationship between threaten and attack will not be captured.

The intellectual weakness the CD approach perpetrates is that one can always "hack" one more sentence. In the above, one can make sure that the lexeme "threaten" and "attack" share a word sense realization. But for a general solution, one needs to rely on the fact that similar meanings are represented in structurally similar manners.

The CD advocates rightfully claim that its "canonical representation of meaning" is one of its strong points. This property is necessary if one is to have any hope of processing the representation in a uniform manner. However, even if CDs provide such canonical representations of meanings, implementations of CD-based systems rely on ad hoc dictionaries. What does the word "canonical" mean if one ends up with an implementation in which vocabulary is defined in terms of tests and actions that "can be any kind of behavior available to the entity doing the comprehending"?

2.4 FUTURE WORK

On balance, then, APE-II tested CD theory and found it wanting. APE's strength is its treatment of lexical ambiguity through selectional restrictions and case grammar. Its weaknesses are due to its lack of a strong linguistic model and to its lack of a fully developed contextual representation scheme. A mature script mechanism, a powerful case grammar, and an explicit linguistic model of English are needed to develop a parser which can accept and understand most user inputs in a restricted domain, explain errors and misunderstandings, and derive the referent of pronouns and noun phrases. CD theory is not enough.

One can regard the experience of the project with APE-II as an experiment. One result of the experiment has been summarized above: CD theory is insufficient. Looked at in a more positive light, however, the APE-II experience provides a host of valuable pointers about directions for future work in natural language.

Interestingly, CD advocates themselves have quietly come to some of the conclusions discussed above. Today, CD based parsers include syntactic analyzers as a matter of course. There is also less attention paid to the issue of "getting the canonical primitives just right." CD research has come a long way from the rather dogmatic early days; most researchers in natural language would agree that there is no point in spending time arguing over the merits of a strictly semantic system versus a more hybrid approach; the most challenging issues are in other areas.

In the first place, future work on a natural language interface should make more use of scripts as a mechanism to define and identify user goals. It is indeed important to give a natural language interface an ability to understand a user's point in asking a question, and scripts are a good place to tie user goals and user questions together. Follow-on work at MITRE will concentrate on this task; without the results provided by the KRS project, we would be at a loss for a starting point.

Another new direction for natural language research is in knowledge representation as it relates to word definition. A more general, less temperamental, and more automated method of defining words must be found. One key to this is to isolate syntactic and morphological information in one place rather than distributing it throughout vocabulary. This will greatly reduce the burden of defining a new word, but it does not do away with the need to find a reliable semantic representation scheme. The search for the latter should be a major priority for researchers engaged in follow-on work.

Finally, researchers must spend more time dealing with the issues raised by portability. Tools to ease the process of moving the interface from one backend to another should become a priority. Such tools should, at a minimum, reduce the time needed to define new words and new scripts.

SECTION 3 - THE KRS ARCHITECTURE

3.0 KRS ARCHITECTURAL ELEMENTS

A number of knowledge-based application programs, such as MYCIN [Shortliffe 76] and a preliminary version of KNOBS, have been constructed as pure production systems. The advantages of such systems, particularly the facileness of explanation and modification as well as the encapsulation of knowledge into discrete nuggets, pertains to the KRS applications. Rules are an effective means to express knowledge about choices of weapons and platforms, for example, since such knowledge is judgmental, and must be easily modified to support tactical improvisation. However, the considerations involved can span several data bases (resources, targets, geography, weather), making the localization of any data item problematic.

In this section, we discuss the basic components of KRS and, in some cases, contrast the KRS implementation to KNOBS. System components carried over from KNOBS unchanged are not discussed. A good description of KNOBS can be found in [Engelman 79] and [Brown 86].

3.1 FRAMES

A set of labeled items to be provided with values is the simplest case of a structure called a frame. The popularity of frames as a knowledge representation technique arose from an article by Minsky [Minsky 75]. Frames generalize the concept of a property list for an object, or a set of attribute-value pairs, as a natural format for storing knowledge about the object. KRS uses frames both for general factual knowledge about airbases, aircraft, etc., and to represent missions.

A number of frame representation languages or data access packages, seven of which are discussed in [Stefik 79], have been developed as LISP extensions. One of these is a system called FRL (for "Frame Representation Language"). Developed at MIT by Roberts and Goldstein, it implements a particular model of frames and provides functions to create and manipulate them [Roberts 77a]. FRL was written in MACLISP and translated into INTERLISP for KNOBS [Ericson 79] and then to Zetalisp for the 3600. Several extensions to FRL have been made, of which the most important is the frame instantiator discussed in the next section.

Among the systems available, FRL was chosen largely because of its architectural simplicity. FRL's capacity to inherit values along semantic pointer paths, and to store and automatically trigger procedures has found considerable use in KNOBS. Nevertheless, one of the other systems, or one created specifically for KNOBS, might very well have been satisfactory, at least as a starting point.

An FRL frame is a nested association list. The frame has up to five levels of embedding, indicated below:

```
(frame
  (slot (facet (datum (label message ... ))
                (datum ... ) ... )
    (facet ... ) ... )
  ... )
  (slot ... )
  ... )
```

What we have been calling an "item" is a "slot" in FRL. The value of a slot is a datum within a SVALUE facet. There are a few other standard facets whose significance is built into the FRL package. Among these, the \$IF-ADDED, \$IF-NEEDED, and \$DEFAULT facets are discussed below. One may put arbitrary facets into a frame, and the standard facets need not all be present.

3.1.1 The Data Base

The data base for the Air Force application has information about 53 tactical units distributed among eight airbases in Germany and Great Britain. There are 600 frames in the target data base. This translates to 303 targets plus their subcomponents, which are currently not considered as separate targets. The data base contains generic information such as the speeds and ranges of 18 types of aircraft and the kinds of radar associated with several types of Soviet surface-to-air missiles (SAMs). The data are plausible, but neither classified nor accurate.

Besides specific data such as the range of a particular type of aircraft or the location of an airbase, the frame system also stores semantic information, showing relationships between objects. In our current application, it is essential to know, for example, that an F-4C is a kind of fighter, which is, in turn, a kind of aircraft. Semantic links between objects of the type "A is a kind of B," are supported in a frame data base by providing frame A with an "a-kind-of" (AKO) slot whose value is the (name of) frame B.

Frames can represent either generic concepts or individuals. The frame TARGET is an example of a generic frame. It has an INSTANCES item with a list of five values: PASSAGE, FACILITY, ELECTRONICS, WEAPON, and VEHICLE. Each of these is the name of a generic frame having an AKO slot with value TARGET. AKO is the relationship inverse to INSTANCES. PASSAGE and the other frames mentioned are also generic; AKO may be thought of as meaning "subclass."

The OCA generic frame has an INDIVIDUALS slot with values such as OCA1001, a current mission being planned by the user. Each mission frame has "an -individual-of" (AIO) slot, with value OCA. AIO is really set membership. (Individuals were referred to as "instances" in [Stefik 79] and [Szolovits 77]).

Other common relationships between KNOBS frames are: "a-part-of" (APO) with the inverse "parts-of-which-are" (POWA) and TEMPLATE with the inverse "a-template -of" (ATO). Many items have values (like numbers) that are not frames.

3.1.2 FRL Functions

FRL provides a collection of functions for creating (DEFRAME, FCREATE, FRAME+), destroying (FERASE, FDESTROY), updating (FPUT, FDELETE, FREMOVE), and retrieving information from frames (FGET, MGET). The ones mentioned are representative. The FRL manual lists 63, and KRS has added new ones. The functions MGET, FPN, FREMOVE, F-REPLACE, and DEFRAME are the main interface functions between KRS and FRL.

The basic KRS function for retrieving information from a frame is MGET. MGET is called with an access path to the part of the frame desired, and returns with a list of the values found. For example, (MGET 'AIRCRAFT 'POWA '\$VALUE) gets (SCL), representing aircraft as having a part called a "standard configuration load."

3.1.3 Inheritance

If MGET is asked for the value under the \$VALUE facet of a slot but there is no value, or the slot itself is not present in the frame, it does not give up. Instead, it looks for a generic frame of which this frame is an instance, listed under the AKO slot, and gets the value from there. As many AKO links will be traversed as necessary until a value is found or no AKO link exists. Thus, a value under any slot can be inherited from a higher-level generic frame.

MGET also recognizes AIO links. If some rule needed to check, for example, what the THREAT-TYPE of target BE50432 was, the system would execute (MGET 'BE50432 'THREAT-TYPE '\$VALUE). Now, BE50432 does not have a THREAT-TYPE slot, but it is AIO SA-3, which is AKO SAM. And the SAM frame has a THREAT-TYPE with value SAM-THREAT. For this sort of search to make sense, only the first link in the chain can be an AIO; the rest have to be AKOs.

If the value is not present and there is no AKO link to follow, MGET looks for a \$DEFAULT facet and takes the value from there.

3.1.4 Procedural Attachment

Besides values and frame names, the data in slots may also be procedures that are activated automatically as a result of FRL actions. \$IF-ADDED and \$IF-REMOVED expressions are evaluated when a datum is added (by FPUT, for example) or removed (by FREMOVE) from the \$VALUE slot. An \$IF-NEEDED facet is the conventional location for an expression intended to be evaluated when MGET fails to find any value, even after following AKOs and looking for a \$DEFAULT.

3.2 FRAME INSTANTIATOR

Filling in the item values for a particular mission is handled in KNOBS by instantiating a template. The template is a frame with the same items as the new mission frame to be created, and serves as a pattern for its slots. The template also contains a list of constraints which apply to the mission's slot values.

While this template information could have been put in the generic frame for each mission type, it is felt that this information is so functionally distinct that a separate frame is more appropriate.

3.2.1 Template Slots

Let us take a closer look at a mission template frame. OCAT, the offensive counter-air mission template, is given partially below. All of its slots are listed, but only some representative facets and data are given.

```

(OCAT
  (AIO ($VALUE (TEMPLATE)))
  (ATO ($VALUE (OCA)))
  (TARGET (SPROMPT ((TARGET-PROMPT :FRAME)))
    (STYPE ((ATOM (LITATOM))))
    ...)
  (PD ...)
  (AIRCRAFT ...)
  (ACNUMBER ...)
  (ORDNANCE ...)
  (REFUELSVC ...)
  (AIRBASE ...)
  (UNIT ...)
  (TOT ...)
  (TD ...)
  (CALL-SIGN ...)
  (TRANSPONDER ...)
  (FREQ ...)
  (ATEND ($VALUE ((BK-SAM-WARN :FRAME))))
  (ATBEGIN ($VALUE ...))
  (CONSTRAINTS ($VALUE (((TGTP1 (TARGET) ...)
    (ACP1 (AIRCRAFT) ...)
    ...))))))
  (SLOT-ORDER ($VALUE ((TARGET PD AIRCRAFT AIRBASE ORDNANCE
    ACNUMBER UNIT TD TOT))))))

```

When KRS is given the command to create an OCA mission, like OCA1001, a new mission frame is created with only an AIO slot pointing to the OCA frame. When the user selects the ADD/REPLACE option for a slot, the desired new value is obtained in a manner determined by evaluating any expression found in the SPROMPT facet under that slot in the template. If the SPROMPT facet is absent (the usual case), a type-in window is provided for the user to input the value.

The input is given a type check to see if it is suitable for the current item; these restrictions are found in the STYPE facet for each template item. A value that has passed its type check is inserted as a datum in SVALUE facet of the mission slot. For some item slots there is a STRANSLATE facet specifying a function used to convert the value from the user's input format to some internal form.

3.2.2 Automatic Slots and Warning

Certain mission menu items like CALL-SIGN have "-Automatic-" beside them. These slots have values that can be calculated in a routine automatable fashion, and are filled in automatically by KNOBS as soon as the items on which they depend are available. A CALL-SIGN, for example, can be assigned as soon as the UNIT and ACNUMBER are filled without error. Automatic slots are filled in by "demons," which are like constraints except that they are executed for their effects and not to test their arguments. A demon reference looks just like an ordinary constraint reference except that "*DEMON*" appears among its 'ags.

Some demons could probably be implemented simply through the use of the SIF-ADDED and SIF-REMOVED facets of the slots that constitute the demon arguments.

3.3 CHOICE GENERATION AND PLANNING

KRS has four facilities to help the planner make choices for mission items, ordered by increasing level of initiative exercised by the system: (1) Constraint checking is performed after the planner proposes a value. (2) Acceptable choices can be enumerated. A choice is "acceptable" if it does not violate any constraint, given the present assignment of item values. (3) After enumeration, an ordering request will order the enumerated choices by preference. (4) The planner can just request "AUTOPLAN" to invoke the AUTOPLAN facility; this finds a consistent set of preferred values for all of the remaining items, or determines that a consistent assignment does not exist. The discussions of autoplanning in this section only deals with the planning of a single mission. KRS also uses strategic-base planning and replanning for multiple missions and this is discussed in section 4.

3.3.1 Constraints

A constraint reference has the form:

(constraint arglist spec flags slots)

Some or all of the items in the arglist are the names of slots in the current mission frame that are involved in the constraint. However, KRS also handles constraints that involve values from several frames needed for example, to coordinate OCA and refueling missions. To obtain a slot in a frame other than the current mission (a remote slot), an access path appears in the arglist instead of just the slot name. For example,

(LOC REFUELSVC STATION ORBIT LOCATION)

references the LOCATION slot in the frame filling the ORBIT slot in the frame found in the STATION slot of the frame filling the REFUELSVC slot at the current frame. This argument will be referred to by the name "LOC" within the "spec".

The "spec" is a specification that indicates when the constraint is "timely"; i.e., when enough of its arguments have values available so that the constraint function can be evaluated. Even when a slot has a value, it may have been involved in the failure of a higher priority constraint, in which case we say that the slot is "trapped" at a higher level. Some constraints can use trapped values, while others should not.

A timeliness specification is a Boolean expression whose terms are slot names (or argument names, in the case of remote slots). The specification

(*AND* TARGET (*OR* UNIT AIRCRAFT))

means that the constraint can be evaluated if the TARGET slot has a value and either the UNIT or AIRCRAFT slot does. In other words, an occurrence of a slot or argument name in a specification is interpreted as TRUE if the slot has a value and the slot is not trapped, FALSE otherwise. Other Boolean connectives may also be used: *NOT*, *XOR*, and *AT-LEAST* (a threshold function).

Although a slot is not ordinarily considered available if it is trapped, the term (*TRAP-IGNORE* slot) in place of "slot" is TRUE if the slot has a value, whether it is trapped or not. (*TRAP-HIDE* slot) is similar, except that the constraint receives NIL in place of the trapped slot value. (*OPT* slot) means that the timeliness of the constraint will be unaffected by the slot having no value, although an *OPT* term is still FALSE if the slot is present and trapped.

Following the spec in the constraint reference is a list of flags. The only flags currently in use are the *DEMON* mentioned above, and (*MASTER* ...) which is discussed below in the subsection on multiple frame co-instantiation.

The slots part contains information relevant to the applicability of a constraint to a slot. Normally, a constraint should be considered (if it is timely) whenever a slot in its arglist has a value change. This can, however, be more complicated, especially when remote slots are involved. A change in a slot that is a link in an access path should trigger a constraint even though that slot is not in the arglist.

The constraint references appear in the CONSTRAINTS slot of a template. A loaded template is converted to its run-time form during initializations. In its simplest form, a constraint reference will contain only the constraint name and arglist, and the remaining elements will be calculated at initialization.

3.3.1.1 Non-Monotonic Logic

A planner in the process of instantiating a mission is interacting with a logical system containing statements about the values of the mission items: has a given value been specified, and has it passed all constraints which can be evaluated and which affect it (given other values are known and accepted)? In normal or "monotonic" logic, values are either good or bad and adding new information can never change a valid conclusion. Thus, the statement set will grow monotonically as new information is added.

KRS takes the point of view that when a constraint is violated, any one of the items involved may be at fault, not necessarily the new one. Adding a value may trigger a constraint which says that not only is the new value in trouble, but so are others which previously had no problems. This means that new information may cause old conclusions to be revised, making our logical system "non-monotonic" [McDermott 78, McDermott 80, Reiter 79].

3.3.2 Enumeration

Note, first of all, that there are two kinds of mission items: "discrete" items with an explicit list of possible values, and "continuous" items that have an infinite choice of possible values. The discrete item AIRCRAFT has only the values F-4C, F-111E, etc., found by inheriting the INSTANCES of the generic AIRCRAFT frame. TD slots are effectively continuous, however, and can be filled with any date-time combination. An enumeration of possible values for a continuous item is expressed in terms of intervals [A, B] from a low value to a high value. If the value is unrestricted above or below, this is indicated by an arbitrary practical bound; for example, there is an "end-of-time" date for in the future from now that serves as an upper bound for all time values.

A simple way to find all acceptable values for an item would be to start with a list of all possible values, and then apply every timely constraint to each possible value. Unfortunately, it is impossible to test all the values in a continuous interval, and even for an explicit list it may be inefficient.

Instead, constraints are "inverted" wherever possible. Inverting a constraint means listing all of the values for an item that are acceptable to that constraint, given all the other arguments of the constraint. A constraint may or may not be invertible for any one of its arguments.

Enumeration, therefore, works like this: the appropriate inversion function of each of the timely constraints is run to get a list or range of values acceptable to that constraint. These lists are intersected to find those values acceptable to all the invertible constraints. If the result is an explicit list, the timely non-invertible constraints are checked to see if any of the values on that list should be eliminated. This is because filters are run on the final intersection of values returned by the generators. It is possible that a value eliminated (by omission) by a generator would also have failed a filter which has a higher priority than the generator. An explanation of this failure should be based on the filter, but additional processing is required to detect this. Having a generator version of every constraint would bypass this problem.

KNOBS automatically sorts the template's constraint references for each slot into a \$GENERATORS facet for inverted constraints, and a \$FILTERS facet for the remaining constraints that could not be inverted efficiently.

As an example, here is the relevant portion of the ORDNANCE slot in the OCA template:

```
(ORDNANCE
...
($GENERATORS
  (((ORD-FROM-P1 ...)
    (ORDNANCE-FROM-USERP1 ...)
    (ORD-FROM-AC-1 ...)
    (ORDNANCE-FROM-UNIT-ORDNANCEP1 ...)
    (ORDNANCE-FROM-ACAVAILP1 ...))))
($FILTERS
  (((TARGORDP1 ...)
    (PDCHKP1 ...))))))
```

The ORDNANCE slot has five generators and two filters. The ORD-FROM-AC-1 generator, for example, inverts the constraint ACORDP1 which tests whether an ordnance is carried by a given aircraft. The generator instead takes only the aircraft as its argument and performs a data base lookup, returning a list of all ordnances that the aircraft can carry. Depending on the situation, enumeration of ORDNANCE would start with a list of all ordnances, remove any not on the restriction list (if the user has restricted ORDNANCE), remove those not carried by the aircraft (if the aircraft is known), discard those not supplied by the unit (if known), and finally remove any that will not be available from the unit in sufficient quantity at the time of the mission (if this can be determined). Any survivors would then be filtered, rejecting those not appropriate for the target type or which do not achieve the desired probability of destruction. In a typical case, some or most of the generators and filters will not be timely because required arguments will not yet be known.

The user can request enumeration of a slot by clicking right with the mouse on that slot to produce a menu of slot operations. Mousing on "Enumerate" will produce a menu of possible values. One of these can then be selected. Alternatively, the user can ask for "acceptable" values for a slot using English input.

3.3.3 Ordering

The ordering option ranks the list of acceptable choices, with some values tied. Not all slots can be ordered. Continuous slots are not orderable, and even discrete slots often cannot be reasonably ordered. However, subranges of continuous slots can be ordered. For example, if a mission is being planned for a certain day but with no other restrictions on time, we might order the TD range into 0600 - 1100, 1400 - 1500, 0700 - 0900, etc. This would tend to create missions that fall within evening, morning and afternoon time periods.

Choices are ordered on the basis of preference rules. The probability of destruction can sometimes be used to break ties. Preference rules are like constraint rules except that their conclusions are of the form:

(SUGGEST-item ?MSN type)

where "item" is a mission item like ORDNANCE and "type" is a generic frame having a slot type of item as INSTANCES or INDIVIDUALS. For example, the rule

```

(AIR-TO-AIR-THREAT ((TARGET ?MSN ?TARG))
  ((AIRBORNE-DEFENSE-ACTIVITY ?TARG ?D)
    (>= ?D 4)))
(SUGGEST-AIRCRAFT ?MSN FIGHTER)
(SUGGEST-ORDNANCE ?MSN AIR-TO-AIR)))

```

suggests that fighters with air-to-air ordnance be used when the enemy fighter defense of the target is substantial. When ordering a list of aircraft, therefore, fighters will be given a higher recommendation than others, and when ordering a list of weapon loads, those that include munitions with air-to-air capability will be preferred. The final choices must, of course, still result in aircraft that are configured to inflict the desired damage on the ground target.

A user's request to order the possible values for a slot is accomplished by selecting "order" from either the slots menu or operations, or from an enumeration menu following enumeration of an orderable slot. A menu will be produced with the preferred values toward the top and equally recommended values sharing the same line. Clicking left or right on a value will then select it for the slot. Clicking middle will produce a line-based explanation of the ranking of that value.

3.4 FROM KNOBS TO KRS

The evolution from KNOBS to KRS included work on constraint references, enumeration, ordering, slot status records, autopanning, constraint diagnostic information, and resource tracking.

3.4.1 Constraint References

There has not been much change in the form of constraint reference as developed in KNOBS and carried over into KRS. However, the development of refueling missions in KNOBS and, consequently, constraints that cross frame boundaries was new and somewhat incomplete. A lot of work went into understanding the code and getting it to work. In particular, the code that initialized templates, completing the constraint reference fields and assigning generators and filters to slots did not always handle access paths properly. For example, it did not realize that a constraint depending on the OCA REFUELSVC and ORB LOCATION slots was applicable to the STN ORBIT slot. Although this slot is not an argument, it is in the path connecting two arguments, and changing its value will change the orbit, and therefore, also the orbit location. The code checking timeliness and retrieving values for slot arguments also had some problems with access paths. In some places, it failed to consider the "MASTER" specification identifying the starting point of the path. Finally, some of the information calculated for template initialization (specifically the /SLOTS/DATA) was often ignored in situations where it already had the precise information desired. Instead, this information was recalculated, again with errors where paths were involved.

3.4.2 Enumeration

Many slots in the refueling frames were not enumerable, and, therefore, could also not be autopanned. Furthermore, slots are now enumerable.

Enumeration employed a questionable method of obtaining an initial set of candidates. It now uses the new generator slot, which always returns a complete set of candidates (the set of all known aircraft, for example).

Slots in the refueling frames were not properly enumerated. They tended to produce either all candidates or the first type of token created blank mission. They now have generators that return all candidates or the first type and return acceptable values and/or a CREATE token. If CREATE is chosen, a new mission of the proper type is created at that time.

A generator now has access to the candidates that have survived all earlier generators. In some cases, generators end up operating in a manner similar to filters by running a test on each value. It is more efficient to limit this test to the remaining candidates instead of all initial instances.

A generator can also return a no change token to indicate that it did not consider itself applicable and that enumeration should proceed with the previous list of remaining candidates. For example, a user-restriction constraint returns the restriction set for a slot and these values are intersected with the previous candidates to obtain the new candidate list. If a slot had no restriction, it had to generate a result that was sure to contain all previous values, so that none would be eliminated in the intersection. It now can simply indicate that there should be no change.

Generators now may return diagnostic information as well as successes and LISPTRAN information.

Enumeration is now better at handling ranges. In some cases, ranges have been converted to discrete values. PD, for example, may in theory be represented by a range between zero and one. It seems reasonable for enumeration to provide a menu of discrete values at 0.05 intervals, allowing the user to make a selection with the mouse. If a user thinks there is a meaningful difference between 0.627 and either 0.60 or 0.65 (and there isn't), then that value may still be input by hand. Even true ranges are represented in popup menus. The user may select either end point or enter a mode that allows selection of intermediate values.

3.4.3 Ordering

Most slots now have ordering functions. Probably the biggest difference is that ranges are now orderable. Previously, ordering was only offered as part of an enumeration menu for discrete values, something not used to display the results of enumerating range-valued slots. More importantly, no attempt was made to write an ordering function for a continuous range. However, it does make sense to break ranges up into an ordered set of subranges. In fact, the enumeration result may already contain many intervals in a range (e.g., a TOT based on various intervals when an aircraft is available). Suppose a TOT can occur at any time on the ATO date. If left unordered, AUTOPLAN would select the first time, creating a mission carried out in the darkness of early morning for no good reason. It makes more sense to recommend a period in mid-morning, followed by mid-afternoon, etc. There are other times when ordering a range might simply consist of reversing it. For example, it might be preferable to have a tanker carry a full load of fuel so that it is available for unexpected services. The amount of fuel would then be an ordered range from the full load down to the amount allocated for services.

3.4.4 History and Slot Status

A record of the origin of each slot's current value and restriction is maintained. This "status" indicates whether the value was input by the user, AUTOPLAN, the strategist, etc. This is particularly important when the strategist is considering changes. User values are often considered more important than system derived values. A separate status is maintained for each value in a multi-valued slot.

The show history agreement shows all the slot value insertions and deletions in a most recent first order. This can be useful in making changes, restoring old values, or returning to a pre-AUTOPLAN state.

3.4.5 Multiargument Constraints, Restrictions, Timeliness and Autoplanning

It is often easy to write a KRS constraint that will return the correct answer once it has known values for all of its parameters. These values normally are found residing in the slots of a mission frame. If a slot referenced by a constraint argument is empty, then the constraint is probably not "timely" and will not contribute to the constraint check being performed. This seems to be a reasonable approach, but it does sometimes lead to situations that are less than desirable.

One problem stems from the fact that the same constraint mechanism is used for both automatic planning and user-controlled slot filling. What may appear adequate and efficient in one mode can be unreasonable and awkward in the other. For example, AUTOPLAN is specifically designed to handle the possibility that a value returned by the enumeration and ordering process will fail to lead to the successful completion of a mission. It is common, however, for a user to be less understanding. If a system-recommended value immediately fails constraints, this is a clear discrepancy between a constraint and its generator (which should be corrected). But, while the user has to go to some trouble to delete the value and try another, the same process during autopanning amounts to little more than a momentary flash on the screen before backtracking occurs and the next value is substituted. A more common situation occurs when a recommended value is accepted, but then no value is found for the next slot enumerated. Again, AUTOPLAN accepts this in stride, while the user wonders why the system failed to realize that the previous value would lead to failure. The answer is that the applicable constraint was not yet timely (since it depended on possible values in the next slot) and, therefore, did not affect the previous choice.

The problem can become worse when constraints depend on a large number of slot values. In this case, the user might plan several slots before encountering failures that are eventually traced back to one of the first slots filled in. The cause of the failure might even be quite obscure and not something that was reasonably predictable at a much earlier step. While this may make the situation more understandable, it has also led to a much greater waste of time than the "dumber" failures discussed previously. It is even possible that the automatic planning mode will start to look bad in this case. If a constraint waits until each of seven different slots has a value before firing, and if many of these have large candidate pools or ranges, then there can be a great amount of trial and error before all combinations are exhausted or a solution is found. The situation becomes even worse when slots have fillers that are themselves missions which also require planning.

A related problem occurs when slots have been restricted. When a slot has a restriction set, it has always been the case that selecting a value not in this set will lead to a violation. Similarly, enumeration of this slot will eliminate any values not in the restriction set. However, a constraint that relies on a slot that is empty but restricted will still not be timely. Consider a constraint that makes sure that an aircraft is found at the selected airbase. If an aircraft is selected when there is no value in the AIRBASE slot, then this constraint does not fire. Suppose that AIRBASE is enumerated next and that there are no acceptable candidates because the only airbases that have the aircraft are not in an existing restriction set. Again, a user would probably prefer that this problem be detected during the earlier enumeration of AIRCRAFT.

It is apparent from these examples that there are times when it would be helpful if constraints and generators were "smarter" and could fire and make correct decisions even when some required information is not explicitly available.

3.4.5.1 Possible Solutions

One solution would be to write constraints that intelligently analyze the situation (both present values and all future possibilities) so that a value is passed only if the mission can be successfully completed for some set of subsequent values. The advantage would be that the user would never be led astray, and even autopanning would never need to back up. Assuming that a simple ordering function could be run on initial candidates, enumeration conducted for AUTOPLAN would only have to return the first successful candidate instead of the entire enumeration set. The disadvantage, of course, is that such an approach is exceedingly complex and expensive. Also, while it is true that a user would rather wait a while to get a correct answer during manual input, long pauses during autopanning are probably less acceptable than watching it at work trying various combinations.

Another approach would be to leave the constraints unchanged but alter the control structure to accomplish the same goal. The simplest way would be to "quietly" autoplan after each user input (or even

for each enumeration value) to ensure that the mission can be completed. There is not necessarily much difference between looking ahead at all combinations and actually going ahead and trying them. In this case, the difference would be in what the system would allow the user to see. Again, the expense of this approach is a main disadvantage. It could, however, be relatively simple to implement.

A possible solution addressing combinatorial explosion during autoplanning would be to retain the old approach but to make it smarter. If useful failure information is returned (and this has been implemented), it would be possible to eliminate many of the candidates. After backing up, AUTOPLAN would avoid trying values that are destined to fail for reasons similar to the last failure. The problem in this case is that failure information becomes less relevant as AUTOPLAN backs up further and further from the failure it pertains to. It might still reduce trial and error to a reasonable level for autoplanning, but it is probably not applicable to the user-controlled mode.

It might also be desirable to have two different modes, one for user input and the other for autoplanning. The previous two approaches discussed, for example, are each directed mainly toward the problems occurring in a single mode.

There are also various ways of dealing with the specific case of restriction sets. The two main ideas are to handle the problem in the control structure (by calling the constraint with each value in the restriction set until one combination succeeds) or by changing constraints to check restrictions (and modifying the timeliness check so that the constraint will fire). It is, however, perhaps a mistake to consider this problem separately. There does not seem to be much difference between an explicit restriction on a slot and an implicit restriction resulting from the current state. In one case, the AIRBASE slot may be restricted to two values. In another case, it may be unrestricted but the aircraft already selected may only be present at two different airbases.

3.4.5.2 Implementation

The need for a change was motivated most by experience with only two different types of constraints. One of these, the itinerary checker, checks that an aircraft can be at the right place at the right time. The time, fuel, and location applicable to the arrival at and departure from each point in the itinerary must be consistent with each other and those for other points. This constraint depends on the TARGET, AIRBASE, AIRCRAFT, UNIT (since this may yield the airbase or aircraft), TD, TOT, and any REFUELSVCs. The refuel services each have additional pertinent values - the service's START-TIME, DURATION, and DISBURSEMENT, and the orbit's LOCATION. The second multi-valued constraint checks resource availability. The aircraft must be available in the type and quantity and at the location required by the mission for the specified time interval. This constraint is, in some way, dependent on every slot in the OCA mission. The AIRCRAFT, ACNUMBER, TD, TOT, UNIT, AIRBASE, TARGET, ORDNANCE, REFUELSVC and PD slots all affect availability by influencing the determination of one or more of the following: aircraft type, number of aircraft needed, total time the aircraft are needed, and source of the aircraft. The slots depend on each other in a somewhat circular fashion. Until the times are known, we cannot know exactly when aircraft are needed or when a refueling rendezvous should occur. But the times will not be known until other details of the trip have been calculated. The trial and error method does not work very well in these cases. There are too many trials and too many errors, particularly if each trial leads into a new set of refueling frames.

No general solution to the backtracking versus look-ahead problem has been developed. Instead, the approach taken is basically to keep backtracking but to avoid it in specific cases. The availability and itinerary constraints have consequently been written so that they fire very early in mission planning and then continue to fire each time an additional value becomes known. They each depend on hypothesizing about possible values of unfilled slots. They are both fairly complex and expensive, have required quite a bit of revision, and remain far less than perfect (although they also do a lot of good work). Writing

generator versions for each of the many slots (which is more efficient than forcing the constraints to serve as filters) greatly increased the amount of code required. Both constraints now require at least the TARGET, AIRBASE, and AIRCRAFT to be known before they will fire. (Currently, a knowledge of the UNIT provides both AIRCRAFT and AIRBASE unambiguously, so the constraints will also fire with just TARGET and UNIT.) The status of all other slots or restriction sets is obviously important to the calculation of the constraint check, but does not affect the timeliness.

The itinerary checker for an OCA mission knows that the mission aircraft undergoes preparation at an airbase, then departs for a target, and finally returns to the airbase, possibly with one or more refuelings enroute. It creates a series of time-location-fuel POINTs, filling in values from applicable slots when they are known. Specifically, the D-AB (depart airbase) POINT has a known location (the airbase), might have a known time (the TD) or a time range (the TD restriction set), and can be assumed to have known fuel (maximum fuel for the configuration) since there is currently no way of (or motive for) specifying any other amount. This point is preceded by PREP-AB, the preflight uploading which takes place at the airbase, but does not have a direct user-specifiable time or fuel amount, although it does have a default duration. It can also be assumed that the preparation starts sometime on or just prior to the day of the ATO execution. A-TARGET has a known location (the target), possibly a known time or range (from the TOT) and an unspecified fuel value. The departure from this POINT, D-TARGET, again has a known location, follows the arrival at the target by a default duration, and has an unknown fuel amount that will ultimately depend on the arrival fuel and the duration over the target. An A-REFUEL POINT might initially not even know the location (for a PENDING refuel service, in which case the most convenient orbit location is hypothesized), may or may not know a service time, and generally will not know the amount of fuel on arrival. The departure D-REFUEL is similar, but can at least assume (as a default) that the aircraft has filled up. The final point is A-AB, the return to the airbase at a time and with a fuel amount dependent on the previous point. The general idea is to create POINTs, fill in known values or time ranges, make sure that each point has a location (calculating orbit locations if not known), and connecting the points in the proper order.

The steps described so far may vary somewhat for different mission types. Once a chain of itinerary POINTs has been formed, it is submitted to a POINT consistency checker. This is the common code used for checking all itineraries, whether for OCA, RFL, AEM, or SSM missions, and should not require much future alteration. It is not concerned with the origin of values or how many different POINTs there are. Although it has some knowledge of default values, it will use them only if its caller has not provided others. The addition of other types of POINTs (checkpoints, rendezvous points, etc.) also has little effect, although some knowledge of default values should be added (such as the knowledge that an aircraft normally does not consume either time or fuel in passing through a check point). The consistency checker makes sure each POINT is consistent with the preceding and following POINTs. As it moves forward, comparing each POINT with the next, a time range may be compressed into a shorter range or even a specific time. Such a change is also propagated back to the start when necessary. The checker is prepared for any combination of types of values: The time may be specified, may be a range, or may be totally unknown; the location is always known; the fuel amount may be known, or more commonly, unknown; additionally, the time duration or fuel use between two points may be known even when their values are unknown. By the time a pair of points is considered, the first point in the pair will have a known location, a known fuel amount, and either a known time or time range. Any inconsistency is detected. Some, such as using more fuel and time than necessary between two points will probably ultimately just result in a warning, assuming that the itinerary remains feasible. Other discrepancies will result in a violation of the constraint.

The sequence of POINTs, with their computed values, make up the itinerary of the mission. Often, it is a hypothesized itinerary, since many of its values have not actually been specified yet. Itineraries resulting from either enumeration or slot filling are stored on the relevant frame. They can often be used

without recalculation by other constraints or by their own generators. For example, the availability constraint always fires after the itinerary has been calculated, and it uses this itinerary to obtain the mission times for which the aircraft must be available. Similarly, during the actual planning of refueling missions, an attempt is made to obtain an orbit located at the location hypothesized by the itinerary constraint.

The availability constraint has been made relatively simple (considering that it has eight generators in addition to the constraint) by tying it to the itinerary constraint. This is a compromise that results in occasional failures in foreseeing future dead ends. A slightly earlier version, which rarely failed, would fire on almost no information and cause long pauses before producing a correct (and sometimes impressive) answer. Besides having to try a large number of combinations, it also had to do a lot of calculating and estimating to determine mission times. The current version fires only when an itinerary is known (TARGET and either UNIT or both AIRBASE and AIRCRAFT are known). Checking the availability of aircraft is easy once the time period is known. The unit and number of aircraft may still be unknown, but the procedure in this case is simply to try all reasonable candidates. These are determined by steps such as checking restriction sets, using only units that are at the airbase and have the aircraft, estimating the number of aircraft from target, ordnance, and PD information when available, or assuming a single aircraft as a default.

Consider the enumeration of AIRBASE. The airbase generator of the itinerary constraint first checks if the UNIT is known. In this case, the airbase must be the one at which the unit is located, and the itinerary will already have been checked and stored. The result will not have changed. Otherwise, the generator calls the itinerary checker for each possible value for the airbase. It uses only airbases that have survived all earlier generators in the enumeration. Each itinerary result is stored for possible later use and the successful values are returned. This particular generator does not operate very differently from a filter. It does, however, have the advantage of firing in its normal sequence while a filter always fires after all generators. This avoids the additional effort of restoring the constraint priority order for purposes of failure explanation. Also, the generator is able to consolidate its answer and return better failure information. If there are still some successful candidates, the availability constraint's airbase generator will be called. It first obtains values for the type and number of aircraft if these slots are empty. Next, it retrieves the itinerary for each successful airbase and uses it and the mission times to check for the availability. Each possible unit at the airbase will be checked. The message sent to the resource tracker might also vary depending on whether the actual mission times are known or only a duration within a certain time interval. Successful values are returned along with failure information for any unsuccessful values.

The handling of the related restriction problem has employed various approaches. First, the above mentioned constraints consider restriction sets when hypothesizing about possible values of empty slots. Another experiment was to rewrite a few relatively simple constraints so that they also checked the restriction set when arguments were unknown. A *RES* operator was added to the timeliness specification in the constraint reference for these constraints. This is used to designate certain slots to be timely if they either have a value or a restriction set. Finally, a more general approach that does not require changes to the constraints was also implemented. The constraint evaluation control structure has been modified to check for restriction sets, to consider these slots timely, and to call a constraint for each value in the restriction set. This has the advantage of not requiring constraints to be rewritten and the disadvantage of not being as smart as a specific constraint can be. Once again, the biggest problem with restrictions surfaces when constraints have many values. Restriction sets seem to be only a subset of the larger problem of hypothesizing values and recognizing implicit restriction sets on slots.

3.4.6 Autoplanning

KRS has an AUTOPLAN capability that automatically completes the planning of a mission which has been partially specified by the user. The basic strategy is an exhaustive tree search which selects a value for a slot and then tries to complete its sub-tree (the remaining slots). If unsuccessful, it backs up and

tries the next value. This approach is not as crude as it may first appear. The candidates for a slot have to pass an enumeration procedure that eliminates values unlikely to succeed. The survivors are then ordered best first, as determined by an ordering function. In most cases, the first candidate is, therefore, the best possible filler for that slot. If this were always true, then backing up would never be necessary. However, the eventual success or failure of a candidate sometimes depends on a large number of factors, many with values that will be obtainable only after additional slots are filled. An enumeration process that would guarantee success would need to consider all future possibilities. This would be quite complicated and expensive, making the directed trial-and-error approach seem superior.

3.4.6.1 Problems in Autoplanning

Until recently, the AUTOPLAN facility used by KRS differed little from that developed for KNOBS. The main changes involved a somewhat awkward add-on that allowed the planning of subordinate missions that filled slots in the autoplanned frame. There remained, however, several problems that demanded attention.

When AUTOPLAN fails to fill a slot, it normally backs up to the previous slot and tries the next value. This procedure has at least two drawbacks. First, the previous slot might not be involved in any way in the failure. The old version was already smart enough to usually recognize this and to continue back to the closest previous slot where a changed value had a chance to bypass the failure. The second problem is that the next value may be obviously (to the user) even worse than the one that just failed. The original AUTOPLAN does not detect this, but, instead, tries each remaining value until one succeeds or none remains. Unfortunately, since the "best" candidates are tried first, subsequent trials usually result in even worse failures. Clearly there is a need to use feedback from the constraint failures in deciding which of the remaining values, if any, should be tried. With such information, AUTOPLAN might, for example, know not to try another ordnance unless it does better than B10 against runways. Similarly, knowing that the last TD caused a 23-minute discrepancy is a lot better than just trying the next value, and avoids the old problem of using a predetermined increment to obtain the next value in a range.

Another problem is in the explanation of AUTOPLAN failure. AUTOPLAN can fail due to values or restrictions previously set by the user, or because required resources are not available. Currently, the only reason that AUTOPLAN can give for failure is (sometimes) to inform the user which user-filled slot should be changed; that is, which slot it would back up to if it were allowed to change values input by the user. The planning system will require a better explanation of what went wrong in order to properly tackle the problem. This reason should be based on the slot failure information in the failed subtrees.

A general lack of flexibility was also apparent in several aspects of autoplanning. First, the old AUTOPLAN decides which slots need planning when it is first called. The decision of whether to plan an optional slot or whether a multi-valued slot requires another filler is often best made after all previous slots have been filled. The decision should be made only when it is actually time to plan that slot, since more information may be available. AUTOPLAN also failed to test the completeness of those slot fillers that are themselves missions and which may require further planning. Second, AUTOPLAN expects to plan a mission to completion using a default slot order. A planning system might want the capability to autoplan only designated slots, in a specified order, during certain phases of planning. Third, AUTOPLAN was written before there was a need to plan subordinate missions (such as refueling) as part of the autoplan. This feature was integrated somewhat awkwardly into the existing control structure. Finally, the trace information of the AUTOPLAN's activity, successes, and failures is not sufficient to support a more intelligent backup procedure.

3.4.6.2 The New AUTOPLAN Algorithm

AUTOPLAN has been substantially rewritten, beginning in January 1985. The approach was to first rewrite the basic AUTOPLAN to correct the flexibility problems addressed above, and in a manner com-

patible with the future processing of diagnostic/explanation information. The next step involved adding code to the control structure to make use of the diagnostic information to choose next values or explain failures. The enumeration and constraint failure code also had to be changed to process any returned diagnostic information. Finally, selected constraints and generators were modified to return the information necessary to trigger much of the new code. Following is a description of the new AUTOPLAN and a diagram of the program flow is shown in Figure 3.1:

1. AUTOPLAN can be passed an optional list of slots to designate which slots to plan and the order in which to plan them. The default is obtained from the frame's template SLOT-ORDER slot. (While there is nothing magic about the default slot order, it should, nonetheless, not be changed without some thought. For example, arbitrarily planning the TD and TOT first would probably impose undesired obstacles on the later planning of AIRBASE and AIRCRAFT.)
2. As in the old AUTOPLAN, it first makes sure that any required slots (those with a NEEDED-FOR-AUTOPLAN attribute) have values and that there is no current constraint violation. If the AUTOPLAN can proceed, it simply involves the planning of all designated slots in the given order.
3. Planning a slot
 - a. If no slot remains, AUTOPLAN has succeeded.
 - b. Otherwise, the slot is first checked to determine if it requires planning. Normally a slot is already complete if it has one or more values present, if it is empty but optional, or if it is automatic. However, if a filler is a mission, this mission must itself be complete. If not, an attempt to AUTOPLAN it to completion is made at this time. Finally (actually initially), if the slot has a SCOMPLETE facet, the function found there is used in place of the standard checks to determine completeness. This is particularly appropriate for multiple-valued slots, since it is probably not correct to make any default assumptions about how many values such a slot needs. If the slot does not require planning, AUTOPLAN plans the next slot (3).
 - c. To plan a slot, AUTOPLAN enumerates the slot and orders its candidates. It then tries the first value (4).
4. Trying a value
 - a. If there is no candidate to try, then the attempt to plan the slot has failed. This failure must be analyzed (5).
 - b. If the next candidate is an incomplete mission, AUTOPLAN attempts to complete it by autopanning before proceeding. If this fails, the next value is tried (4).
 - c. The candidate is inserted into the slot and constraints are run. AUTOPLAN then tries to plan the next slot (3). However, if constraints fail (due to a generator/constraint discrepancy),

AUTOPLAN complains but is able to recover and eventually try the next value (4).

5. Analyzing a slot failure

- a. AUTOPLAN analyzes any failed values (from enumeration or actual trials) and retrieves any diagnostics which it stores on the relevant slots in the AUTOPLAN trace for later use.
- b. It also determines what slots are involved in the failures and stores this information as well.
- c. If there is no previous slot to back up to, AUTOPLAN has failed (7).
- d. Otherwise, the current value in the previous slot must be changed (6).

6. Processing a value failure

- a. The failed value is removed from the slot.
- b. Information from 5a & b is used to possibly eliminate some of the remaining slot candidates. The diagnostic information is used to eliminate values which would fail for reasons similar to a previous failure. If the slot is not relevant to any failure, then all candidates are eliminated. An attempt is then made to try another value (4).

7. AUTOPLAN failure

- a. There may be applicable diagnostic information that will allow for a recovery from the failure. Such an attempt may return a new and successful AUTOPLAN, which is returned as the result of the original AUTOPLAN invocation.
- b. Otherwise, AUTOPLAN again runs any applicable diagnostics, this time in a mode which prints messages suggesting steps for the user to take. The trace of the failed AUTOPLAN is returned.

Diagnostic information is an optional addition to the result of constraint evaluation. If it is present and applicable to the situation, it may enhance the planning operation; if it is absent, the system continues to operate as before. AUTOPLAN saves diagnostic information on its planning trace and has the potential to use it in three different situations.

When an AUTOPLAN fails, appropriate diagnostics are evaluated and these print out messages explaining the overall failure with an attempt at emphasizing possible corrective actions for the user to take (instead of just what went wrong).

When a slot value fails (usually because the next slot had no candidates or has exhausted them), the diagnostic is again evaluated, but this time the ordered list of remaining candidates is added as the final optional argument. The diagnostic function acts as a filter which returns only those candidates that pass a test. It is designed to eliminate those values that will probably fail for reasons similar to those of previously failed values.

Finally, when AUTOPLAN has initially appeared to have failed, some diagnostics provide a final chance for success. While most diagnostics are applicable to the failure of a slot in the mission frame, these diagnostics apply to the failure of the overall AUTOPLAN. In such a case, any such "second chance" diagnostics are called. The idea is to provide for a recovery from the AUTOPLAN failure. If any such diagnostic succeeds, its result is returned as the AUTOPLAN result; otherwise, diagnostics are called to explain the failure as described above. The only current example handles refueling. If planning of a mission has failed due to insufficient refueling, an attempt is made to replan the mission with an additional refuel service. If successful, this becomes the result of the AUTOPLAN.

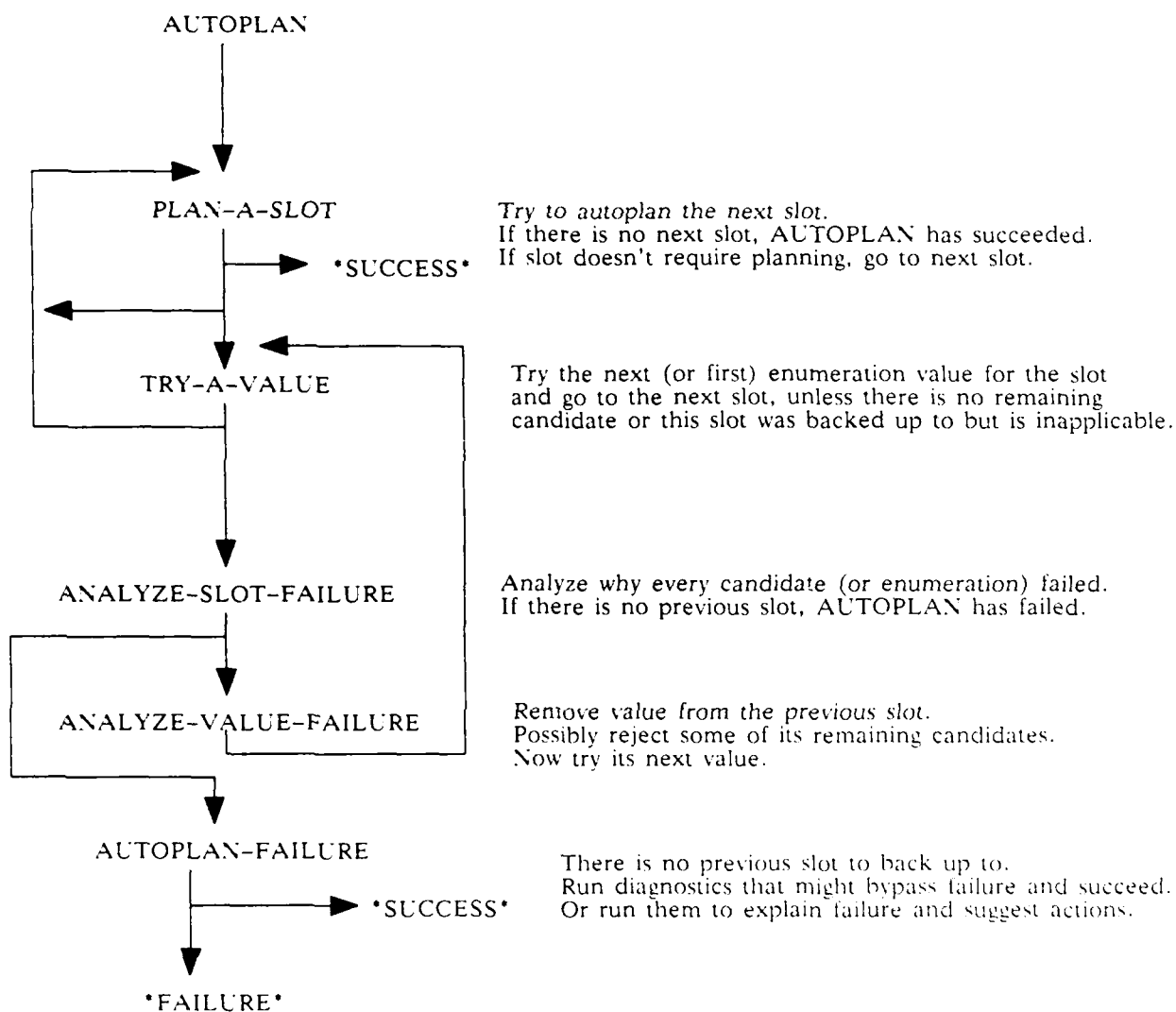


Figure 3.1 AUTOPLAN Algorithm

3.4.6.3 Remaining AUTOPLAN Problems

Most enumeration failures are due to omission from a generated list. Generators are designed to produce successful values as opposed to rejecting values for specific reasons. Often, the manner in which values are generated provides little information as to why some other value failed to be included.

The new AUTOPLAN currently is best at analyzing an enumeration failure of the next slot in order to determine the next value to try. Consider autopanning slots A, B, and C of a mission. Candidates are found for slot A and one is selected; slot B is also found to have candidates; but after filling slot B, the enumeration of slot C fails. Information returned by the enumeration is available on the trace and can be used to analyze the chances of the remaining candidates of slot B. So far, this appears to work satisfactorily. The new algorithm has shown its ability to skip over several candidates to select one that works, or to recognize that all will fail for similar reasons. A more difficult problem occurs, however, when all the candidate values for slot B eventually fail. This means that AUTOPLAN needs to back up to slot A and analyze its remaining values. Information on the trace provides a (possibly different) reason for the failure of each value in slot B. There are probably also values that failed the enumeration of slot B, along with the reasons. One problem is the abundance of information and selecting the right approach. The more serious problem is that most of the information is concerned with slot B and is based on the value in slot A. But this value has now failed and its removal and replacement may lead to a very different situation that might largely invalidate the information returned from the failed subtree.

Messages explaining the overall AUTOPLAN have the same problem. They are basically diagnostics propagated up from a failure that occurred for a specific combination of values in lower slots. They may not seem very relevant once AUTOPLAN has backed up to the starting point. On the other hand, they do identify a specific problem which, if it were bypassed, would allow AUTOPLAN to proceed beyond a previous dead end.

3.4.6.4 Autopanning the REFUELSVC Slot

The REFUELSVC slot has always been a special problem. Sometimes it is not needed at all. When it is needed for a certain combination, we usually want it to initially fail in order to continue to look for a combination that does not require it. Sometimes more than one refueling will be required. Refueling can occur before or after the target, or both. Sometimes we want refueling to be HYPOTHETICAL. In this case, the OCA is planned based on the assumption of one or more refuelings at a convenient time and place. The actual planning of services, choice of orbits, availability of tankers, etc. are matters that can be left to a later time. This may allow for some later optimization of orbit location and tanker usage. On the other hand, any changes from the hypothesized values will have to be corrected (e.g., by adjusting a time in the OCA by a few minutes if the orbit is a little further away).

The constraint that checks that an aircraft can be at the right place at the right time and have enough fuel always considers what is in the REFUELSVC slot. If there is an actual service, it will use the values to determine the time and place of the service, make sure that all values are consistent with the mission (the aircraft can get there at the right time and will receive the correct amount of fuel) and also hypothesize any missing values if the service is incomplete. If it finds a PENDING or HYPOTHETICAL, it handles each in the same way as a totally unplanned service, hypothesizing all values. If the aircraft can make the trip using these services (if any), the constraint passes. When AUTOPLAN finally gets to the REFUELSVC slot it will make sure any service mission is totally complete, it will substitute PENDING with a service that it creates and autopans, and it will leave the HYPOTHETICAL untouched. This is all actually performed as a part of the slot's special completeness test which is conducted before it starts to plan the slot. If successful, this check reports that the slot is already complete.

Assuming that the user has not inserted a value, REFUELSVC will initially be empty and AUTOPLAN will try to plan the mission without refueling. If this is impossible, it will fail before it ever gets to the

REFUELSVC slot, based on the mission's requirements. If it does reach the slot, it is because refueling is not needed, and the completeness checks on the slot does not require planning. If refueling is required, AUTOPLAN will not be able to do it itself. A diagnostic will recognize that some combination of values during the AUTOPLAN trace led to a lack of refueling. It will then take action that might lead to a new AUTOPLAN trace, or service in the REFUELSVC slot. It is worth discussing two alternative methods of achieving this goal.

The diagnostic could store the current refueling time. The next iteration of planning, including either PENDING or HYPOTHETICAL, could restore the contents of service as well as AUTOPLAN. If this succeeds, the new instance of AUTOPLAN receives the result of the original AUTOPLAN. If this AUTOPLAN also fails to refuel, additional services can be added recursively. The problem with this approach is that repeating the AUTOPLAN totally means recalculating lists of combinations which will fail for the same to most reasons as before. Additionally, many combinations of airbase, target, and aircraft will be tried with the additional refueling when the aircraft is perfectly capable of flying the route without extra fuel. This may cause some unwanted problems or warning messages. Finally, after all this work, the combination that needed the refueling may still fail for some other reason (e.g., availability), which is checked after the movement constraint so that it can use the mission time values calculated by it.

In the chosen implementation, the diagnostic follows the AUTOPLAN trace down to each specific failure that was caused by lack of fuel. It restores this situation, adds a HYPOTHETICAL refuel, and recalls AUTOPLAN. This may also reinvok the same diagnostic to add additional refuelings. This is all done automatically and fairly quickly. If it fails to find a solution, the AUTOPLAN is restored to its original values and the user is never bothered. If it succeeds with the additional refuel, the user is asked if this is OK. AUTOPLAN can be told to complete the refuel (replace hypotheticals with services and complete them), accept plan with hypothetical (finished), reject refueling (undo result of autoplan and quit), or reject the particular plan but allow AUTOPLAN to try it for another combination. When the user is asked this question, he can also make the answer effective for subsequent refuel permission requests, so that AUTOPLAN will do it without bothering him any more.

3.4.6.5 Other Autoplan Changes

All types of missions can now be autoplaned. This includes the refueling frames and the new PKG, AEM, and SSM frames. Subordinate missions are automatically planned as part of the AUTOPLAN of slots that need them. In these cases, enumeration can accept existing frames which satisfy constraints for the slot and/or a new instance of the frame can be created. Since incompletely planned missions (either candidates or previous fillers) are automatically autoplaned to completion, a successful AUTOPLAN guarantees a result that is complete throughout all levels. For example, an OCA with only a target can be autoplaned. If a refuel service is required, it creates and plans a SVC mission. The SVC requires a STN to fill its station slot. This could be satisfied by an existing STN which already happens to have a tanker (with sufficient fuel and in a convenient orbit) that can refuel the OCA mission at an acceptable time. Otherwise, a STN would be created. It would have to find or create an acceptable ORB for its orbit and an RFL tanker flight for its refuel slot. New frames (package, air escort, SAM Suppress Missions (SSM)) are also autoplanable.

AUTOPLAN offers selectable display options. AUTOPLAN may involve planning a single frame or planning an entire ATO of many packages, each containing OCA, AEM, and SSM missions, many requiring refueling. In the first case, it may be desirable to AUTOPLAN with the mission window exposed and with detailed information concerning slot candidates under consideration. In the second case, a one line printout, showing each mission and its values, provides a more manageable overview of the ATO planning.

AUTOPLAN messages and the exposure of mission display windows can be controlled through user selection or by certain parts of the code. The planner, for example, occasionally tests a possibility by

autoplanning quietly without any output. The same idea has been applied, to a limited extent, to other KRS messages and output. Different time formats, for example, may be selected (military time format, absence or presence of date or seconds, etc).

3.4.7 Resource Tracking

In KRS, it is feasible to plan an OCA strike against any target in the data base. Failure to successfully plan a mission to completion occurs only when slot values provided by the user impose unsolvable constraints on the planning. For example, the user may refuse to allow a required refueling or might specify an ordnance and an airbase when no aircraft stationed at the airbase carries the ordnance. A more interesting problem occurs when required resources are not available. This situation can also be traced back to the user. Resources will ultimately become available, so shortages occur only with respect to specific time intervals. A mission could still be supported, but only if times are changed, the use of other more available resources is permitted, or competing missions are altered. In the real world, these solutions can each present various difficult problems. The availability of resources is, consequently, one of the most important considerations in planning missions. KRS requires a sophisticated resource availability tracker to support the checking of availability constraints and the planning of missions that bypass some of the problems involved in planning with limited resources.

KNOBS (or at least early KRS) did, in fact, record the usage of aircraft at each unit and check that a mission could be supported. This capability was, however, implemented for a resource rich environment that emphasized the planning of just a few missions. The calculation of mission times (particularly refueled missions) was also not well developed. Resource availability was mainly a matter of keeping records and recommending the use of units that were infrequently tasked. KRS must address the more realistic problem of planning an ATO with many missions and usually with too many high priority targets, too few resources, and insufficient time.

The KRS resource tracker is implemented employing Zetalisp flavors. It currently makes use of two types of objects, the CLASS-TRACKER and the SOURCE-TRACKER, each having associated instance variables and many operations. The SOURCE-TRACKERS track the use of a resource at its actual source (e.g., the 15 F-4Cs assigned to the 110TFW). This is the level at which the actual resources are found. The CLASS-TRACKERS are concerned with various groupings of resources or classes of resources (e.g., all F-4Cs, all OCA aircraft, all aircraft). The top-level *RESOURCE-TRACKER* is simply a CLASS-TRACKER that contains all classes of all types of resources. Most resource queries and operations are initiated by messages sent to this top-level tracker. These, in turn, may be passed to appropriate CLASS-TRACKERS or SOURCE-TRACKERS. Changes to the state of the resource tracker are made through the creation of USAGES. A USAGE consists of the allocation of a quantity of a resource from a specified SOURCE-TRACKER for a certain time interval to support a consumer (e.g., 4 F-4Cs from the 110TFW in support of OCA1003 from 1000 to 1315). A gain (negative quantity) or loss of aircraft involves a usage from the reported time to eternity. Since resource queries are much more numerous than actual changes, the resource tracker maintains various values to avoid constant recomputation. If it becomes necessary to track individual aircraft, then an ITEM-TRACKER could be created for this purpose. At a minimum, it could simply consist of the individual item designation, its SOURCE-TRACKER, and a list of usages that involve the item. Each SOURCE-TRACKER would also maintain a list of its ITEM-TRACKERS.

The CLASS-TRACKER has the following instance variables:

CLASS	- name of the class (e.g., F-4C, or OCA-AIRCRAFT)
INFERIORS	- CLASS-TRACKERS for the immediate subclasses of this tracker
SUPERIORS	- CLASS-TRACKERS that have this tracker as an INFERIORS
SOURCES	- all the SOURCE-TRACKERS of this tracker and all its subclasses
ASSIGNMENT	- total number of end-item resources grouped in this tracker

MAX-1-SOURCE - maximum number assigned to a single SOURCE-TRACKER
 AVAILABILITY - number of end-items available by time

The SOURCE-TRACKER has the following instance variables:

RESOURCE - type of resource (e.g., F-4C)
 ORGANIZATION - unit that provides and controls it (e.g., 110TFW)
 LOCATION - location of the ORGANIZATION (e.g., Hahn)
 USAGE-LIST - list of USAGES of the SOURCE-TRACKER's resource
 ASSIGNMENT - total number assigned
 AVAILABILITY - number available by time
 SUPERIORS - the lowest CLASS-TRACKERS of this resource (e.g., the CLASS-TRACKER for F-4C) (provided only as a pointer)

Currently, about 75 messages are supported by the resource tracker with most of these belonging to the CLASS-TRACKER. For example, the resource tracker can be queried about the availability of F-4Cs as a class, or for the availability of F-4Cs at the 110TFW. The latter provides an answer by sending a request for the availability of resources to the 110TFW-F-4C SOURCE-TRACKER. A direct request by the user is usually not made, since it would normally first require sending a message to ascertain the SOURCE-TRACKER. Similarly, the request about all F-4Cs usually goes to the ALL-RESOURCES CLASS-TRACKER (which is known globally) from which it is relayed to the F-4C tracker. The resource tracker is currently implemented only for aircraft, the most important of KRS resources. Other types of resources can be added without modifying the basic tracker code. The top-level *RESOURCE-TRACKER* (the class of ALL-RESOURCES) would then have inferior classes ALL-AIRCRAFT, ALL-ORDINANCES, etc.

The following is a list of some (anglicized) messages recognized by the resource tracker:

Allocate or deallocate the following resource USAGE (since a mission has just been planned or changed)
 Create a new CLASS-TRACKER and insert it between the following current CLASS-TRACKERS (normally during tracker initialization only).
 What is the availability of KC-135s? (Is lots of refueling feasible?)
 How many F-111Fs are available from 113TFW from 1145 to 1600 ignoring usages by OCA1001 (considering moving OCA1001 from 1430-1645 to the new time)?
 What units have 5 F-111Es available for a 5-hour period between 0600 and 1500, ignoring any USAGE by OCA1001 and OCA1003 (since they could be cancelled)?
 What units have the most F-4Ds available between 1200 and 1500, and how many?
 What units will have at least 3 F-111Fs available for 5 hours, and for what times?
 Will 110TFW have 4 F-4Cs available for 4 hours between 0500 and 1700?
 Does the currently planned allocation of F-4Cs from 111TFW exceed the support capabilities of that unit (now that planes have been reported lost)?
 When will 3 F-111Es be available from 101TFW [for at least 6 hours]?
 What missions are scheduled to use F-4Cs [from 110TFW]?
 What is the greatest number of F-4Cs ever available from any unit (the user has specified an OCA with 5000 F-4Cs)?

SECTION 4 - THE PLANNING COMPONENT

4.0 INTRODUCTION

Although planning has been the subject of AI research for decades, most implementations of planning theories have been small scale laboratory exercises. KNOBS and KRS are unusual because they represent realistic Air Force problems. Because of this, their developers cannot avoid critical planning issues like truth maintenance, consistency checking, goal interaction, combinatorial complexity, non-local effects, and nondeterminism. Unfortunately, no planning theory to date has satisfactorily resolved all of these problems, so MITRE developers have had to rely on a combination of theory, heuristics, and sophisticated workarounds. A recurring theme in the history of KNOBS and KRS has been the search for better, more powerful, planning theories; this is particularly true of KRS because KRS is a much more realistic approach to the mission planning problem than is KNOBS. The latter dealt with planning single isolated missions, a limitation which made a simple, constraint based, bottom up approach possible. With KRS, however, the problem is so complex that a bottom up approach would result in a fatally large search space. As a result, KRS researchers knew that they had to search for a theory which took a top down, strategy oriented approach in which the planner would have some ability to reason about plans and to limit the search through a vast space of possible plans.

In the course of developing a solution to KRS's planning problems, MITRE investigated several contemporary theories, two of which were implemented on an experimental basis. The bulk of this chapter will be devoted to a discussion of these two theories, the theory of fuzzy algorithms as developed by Goldkind and Wilensky's theory of metaplanning. We will first discuss the theories in detail and explain why MITRE considered them to be strong candidates for KRS. In the case of fuzzy algorithms, which were implemented and delivered as part of KRS, we will describe the details of implementation. However, neither Wilensky's theory nor Goldkind's provided an extensible, robust solution to all of KRS's problems, and we will describe the reasons the KRS staff came to believe that the replanning and multi-mission planning problems required a new approach. Finally, we will show how MITRE's experience with KRS led to the development of such a new approach: the AMPS theory of metaplanning. The following chapter, which describes the history of planning at MITRE from an evolutionary perspective, will provide a more detailed account of this theory.

4.1 WILENSKY'S THEORY OF METAPLANNING

In 1983, Wilensky published *Planning and Understanding*, a thorough presentation of his ideas about planning and language understanding. In this work, he not only outlined a theory of planning but also discussed several programs, FAUSTUS, PAM, and PEARL, which were implemented by his students and which embodied aspects of these theories.

The essence of Wilensky's theory is that in order to solve the planning problem one needs a general mechanism to handle both planning and the interaction among plans. In other words, Wilensky claims, it is important to use the same mechanism not only to create a plan but also to recognize a plan, to resolve a conflict among plans, and to choose one plan over another. The reason for this is that plans typically have to deal with not just a single goal but with a variety of subgoals. Subgoals inevitably interact, sometimes contradicting each other, sometimes supporting each other, sometimes forcing the planner to impose a hierarchy on their solutions. Even when goals do not contradict each other, the plans that achieve them may interfere with each other. The questions that arise

can the effects of achieving a goal be predicted?
can the effects of a sequence of action be found by symbolic execution
(e.g., simulation)
even if effects cannot be found by simulation, can potential effects be
anticipated?

In general, the answers to all three questions is "no." Thus, Wilensky and others insist that a regime of interleaved planning and execution is necessary. Furthermore, without some way of thinking about the plans which are used to solve interacting subgoals, the planner is helpless to solve such problems as goal conflict. By using the same mechanism to reach goals and to solve problems caused by interacting subgoals, the theory captures the generalization that planning and manipulating already created plans are the same problem; one can think about the interaction amongst plans in the same way that one can think about goals and states when one is attempting to create a plan.

Wilensky's theory was developed to deal with such problems as replanning and resolving goal conflicts. His solution was to propose another layer for the planning process, the metaplanning layer. Whereas plans dealt with goals and states, metaplans were plans that called on or manipulated other plans. Metagoals were goals that were concerned with the interactions amongst other goals; an example of a metagoal would be *resolve-goal-conflict*. Metathemes were general principles that were applicable to all planning domains. Wilensky identified four *metathemes*, or general principles, which offered guidance in all planning situations. These metathemes were:

1. Don't waste resources
2. Achieve as many goals as possible
3. Maximize the value of the goals achieved
4. Avoid impossible goals

Metathemes could be used to provide guidance for choosing one plan over another. For example, when one plan is wasteful and another economical, the metatheme "don't-waste-resources" gives the planner a reason for choosing the latter over the former. Metaplans enabled the planner to respond to metathemes and to situations in which goals could not be carried out using normal plans. An example of a metaplan might be "try-alternative-plan", another would be "substitute-one-goal-for-another." Plans, goals, and states were represented according to a CD-like knowledge representation scheme; a planning program would have many goals, plans, and states defined in a plan database.

Wilensky proposed a four part design of a planning program in which metaplans and plans were used to carry out goals and metagoals. The four components were:

1. a goal detector to detect the appearance of situations relevant to the planner; in other words, situations in which goals arose.
2. a plan proposer to find stored plans that matched the goals detected by the goal detector. Plans and goals were represented by a CD-like knowledge representation scheme.
3. a projector to simulate the execution of the proposed plans. This is a key component.
4. an executor to carry out the chosen plan.

This proposed program would not only be able to produce plans, it would also be able to recognize them. This is the reason Wilensky entitled his book *Planning and Understanding*; he believed that his system could be applied to natural language understanding as well as to planning, because in-depth understanding requires some ability to recognize plans and goals. This was an important reason the KRS staff chose to investigate this theory. They hoped to use it both to plan and to help APE, KRS's natural language front end, understand user goals in issuing commands and questions.

4.1.1 MITRE Implementation

The KRS staff attempted to implement Wilensky's design on a small testbed world, the world of BOB. In this world, a character named Bob wanted to get rich. He attempted to achieve this goal by winning a fortune at gambling. This scheme spawned the subgoal of going to Atlantic City, which in turn spawned the subgoal of getting the money to get to Atlantic City. The BOB program attempted to manipulate stored plans for achieving these goals and subgoals according to Wilensky's design.

The staff defined the data structures necessary to describe the states of BOB's world as well as the required plans and goals. This required a sizeable amount of work since it was not an easy task to create CD definitions for all the relevant objects. Next, the KRS staff implemented the goal detector and plan proposer, both of which were basically pattern matchers that recognized CD's corresponding to plans and goals. Finally, they created the projector and the executor; at this point, they encountered serious trouble.

The problems encountered in the implementation of BOB derived from two weaknesses with the theory described in *Planning and Understanding*. The first problem was due to a theoretical/implementation gap: Wilensky put the burden of dealing with consistency checking in the proposing and execution of plans entirely on the projector. It was the projector's job to simulate the execution of plans and to keep track of how the world changed as a result of plan application. Wilensky claimed that plans could be simulated deterministically, so that issues of incomplete or missing information about the planner's world could be finessed. However, he said remarkably little about how this critical component, the simulator, actually worked. In fact, the limited description of the simulator's design leads one to believe that this component was not fully implemented at the time *Planning and Understanding* was written. Wilensky himself hints at this in passages like the following:

The difficult problems in conducting a simulation involve reasoning about "possible world" situations which are not amenable to standard temporal logic. Rather than employ a logic for possible worlds, this issue is finessed here by defining hypothetical states in terms of what the planner thinks of in the course of plan construction. In other words, the solution is to let the system assert the changes that would be made into a hypothetical database, in the meantime letting the Goal Detector have access to these states.

Since Wilensky never offered a detailed explanation of how plan execution effected the world, the reader runs a risk of believing that either side effects are not important or that they can easily be predicted and tracked. This is, of course, not true in any realistic problem, but Wilensky was able to "finesse" them in the implementations of small scale planners described in his book.

Unfortunately, KRS did not present a small scale problem, and the lack of an effective truth maintenance system to handle inconsistencies and to track a change would have proved fatal. Even BOB, a greatly scaled down planning problem, proved to have terrible problems in this respect: when the projector was simulating plans while the proposer was creating a new plan, there was no way to prevent the creation of fatal inconsistencies in the BOB world. Furthermore, a look to the future showed that any realistic use

of KRS would someday require an ability to respond to incomplete information, not only about the state of the world but also about the potential outcome of plans. Wilensky's theory did not come to grips with the problem of nondeterminism in planning and offered no guidance in this area.

It should be noted here that there was a way out of the truth maintenance trap, but it would have meant abandoning much of Wilensky's design. In BOB, the need for truth maintenance was primarily for keeping track of situations in simulation. If the need for simulation could have been removed, the need for truth maintenance would have changed its character. However, abandoning simulation would not have worked in Wilensky's design because of the problems described above with prediction the effects of actions. As we will show in the next chapter, MITRE eventually developed a new design, the AMPS design, in which it is possible to make such predictions. In KRS, however, there was no way around these problems.

Complications worsened because of an implementational gap: a critical element of the proposer, the **Make-attribution** routine, was never fully implemented by Wilensky's colleagues, nor fully described in the text of *Planning and Understanding*. **Make-attribution's** job was to identify what was salient or important about a plan; without it, there was no way to distinguish one aspect of a plan from any other. The example Wilensky gives is that of a plan to get a newspaper on a rainy morning. The important thing about this plan is that it results in someone getting wet – an undesirable effect. It was the **Make-attribution** routine's job to notice that the salient feature of the plan was that it resulted in this undesirable state. However, Wilensky admitted that in PANDORA, his own implementation of his theory, this routine was replaced by a special rule in the plan interpreter; this was certainly no help to the MITRE developers. In fact, although Wilensky presented his theory as if it had been fully implemented, it is unclear that this was the case. Neither FAUSTUS nor PANDORA was a fully developed program as he says himself in *Planning and Understanding*, and the various theoretical and implementational gaps described in the preceding paragraphs indicate that *the theory could not be implemented exactly as described*.

Lacking either a truth maintenance scheme or a way to identify salience, the BOB program floundered. It was possible to coax it into carrying out simple plans but only with a great deal of special case programming. After one year of effort, the staff decided to reevaluate the whole attempt and concluded that implementing Wilensky's design in the context of KRS would be a mistake.

This decision was founded on two considerations: there was no proof, judging from the BOB experiment, that the salience and inconsistency problems could ever be resolved, and, in addition, the effort of imposing Wilensky's design on KRS would involve a massive effort amounting to rewriting an estimated 60% of the existing system. For Wilensky's design to work, the entire FRL database along with the constraint mechanism needed to be replaced with CD definitions of the actions, states, and events that would give rise to planning goals. Otherwise, there would be nothing for the goal detector and the other components of the system to recognize. KRS has 75 constraints, 91 generators, 26 ordering guidelines, and almost 1500 frames; redefining these structures in terms of CD could have taken years (see chapter 2 for a discussion of the difficulties involved in creating CD definitions). All of this work would have been in addition to the effort of writing the plans and goals themselves. With no solid evidence that any of this work would produce an effective planner, indeed with experience to the contrary, the KRS staff made a decision to find another approach to planning that could be more easily integrated with KRS.

None of this argues against the intrinsic merits of metaplanning. MITRE remained convinced that metaplanning, the ability to reason about plans, was essential to the solution of the KRS problem. It was Wilensky's implementation, as presented in *Planning and Understanding*, that was eventually rejected by the staff. The KRS developers remained convinced of the importance and value of Wilensky's work, but their experience with his theory indicated that it needed some crucial additions, most notably a truth maintenance system of some kind. They also were forced to conclude that grafting this theory on top of

the bottom up, constraint based KRS approach to planning would take far too long to accomplish within the three years of the program. The amount of work involved would have been equivalent to rewriting the entire program.

In searching for an alternative to Wilensky's theories, MITRE turned to the theory of fuzzy algorithms developed by Goldkind and his colleagues. This is the solution that was eventually implemented in KRS and delivered with the final product. In this implementation, metapanning concepts, called strategies, are used to guide planning. Like Wilensky, the developers of this theory reject a rigid, bottom up, STRIPS-like approach to planning in which plans are broken down into a sequence of operators and arguments in favor of a top down approach that emphasizes an ability to reason about problems and their solutions.

4.2 STRATEGIES AND FUZZY ALGORITHMS

MITRE turned to fuzzy algorithms because they are an attempt to deal with exactly the issues of nondeterminism and combinatorial complexity that make the KRS planning problem so difficult. A Fuzzy Algorithm (FA) consists of a hierarchy of fuzzy plan schemas determined by the heuristics for instantiation accompanying each schema in the hierarchy. The word fuzzy is used to describe this kind of algorithm because in an FA, some steps of the algorithm are left undefined or underspecified. Plan schemas determine a set of possible actions each of which is a particular instantiation of the plan schema. The choice of the actual instantiation that is made in the construction of the plan is determined by heuristics that are sensitive to the environment in which the plan is to be executed. The following sections briefly review the concepts of "epistemological nondeterminism," "strategy vs. tactics," "fuzzy concepts," and "fuzzy algorithms." Fuzzy algorithms, by incorporating fuzzy concept and strategy into the planning process, should be able to deal with many of the problems posed by epistemological nondeterminism, combinatorial complexity, and subgoal interactions. The concept of a fuzzy algorithm unifies several recently but separately proposed solutions to these problems, namely: interleaving planning and execution [McDermott 78; Sacerdoti 79], meta-planning [Wilensky 83], and planning to acquire knowledge [Haas 82]. In addition, fuzzy algorithms provide a method for giving a more central role to the use of global strategies in the planning process.

STRIPS-like (SL) systems assume that a sequence of actions can be found which is known (in advance) to define a sequence of states leading from the initial state to some state in which the goal is realized. However, this assumption is justified only for limited and completely deterministic domains. In order to generate the search space (and find a solution path), it is necessary to know what the effects of each action will be. In SL terminology, we must be able to give the add and delete lists for each operator. Unfortunately, in the real world, things are not this simple; many domains, including that of KRS, exhibit one or more types of what we refer to as "epistemological nondeterminism." (See [Goldkind 83] for a more complete account of this concept.) We often do not know what the effects of an action will be, we also do not know whether the effects produced by the action will persist until the next action in the sequence is performed by the system. Thus, it can be impossible to calculate a path through the search space.

Ordinarily we speak of a world as being "deterministic" if every effect is inevitably predetermined by its cause(s), and as "nondeterministic" if this is not the case. The problem with the real world is that no one knows whether determinism holds or not. Even if determinism is true we cannot know it because no one can be aware of every cause and every effect, nor of every connection between them. As far as the state of our knowledge is concerned, there are many cases in the real world where determinism might be false.

Proposed solutions to the problem of nondeterminism involve interleaving planning and execution [Sacerdoti 79, McDermott 78], planning to acquire knowledge (i.e., knowledge necessary for further planning) [Haas 82], and meta-planning [Wilensky 81a, 81b]. As is apparent from the above discussion

of metaplanning, MITRE saw much value in these approaches and looked for a way to unify them. This attempt led to the implementation of strategies in KRS.

4.2.2 Strategy vs. Tactics

"Strategy" is an overworked word, with no definite agreement on meaning, even among military writers [Wylie 67]. We begin by explaining how the two terms, "strategy," and "tactics," are used in this report.

Strategy is concerned with the global aspects of a problem, tactics with specific details. Often there is no clear dividing line between the two areas; tactics may be the implementation of a strategy in a particular case. Perhaps the clearest way to distinguish between the two is to consider cases where the two conflict.

Tactics tell us how best to accomplish the particular limited goal presently at hand. (In military parlance the proper use of a particular weapon is often referred to as "tactics" [Palmer, Richard 75]). In terms of the problem reduction paradigm, we can regard tactics as a concern with how best to accomplish a particular subgoal in isolation. Strategy, on the other hand, is concerned with the relations among (sub)goals and how these interactions relate to the final objective (or main goal). If we accomplish a particular subgoal at the expense of the main objective, then we may have followed the dictates of tactics with respect to the subgoal and violated the maxims of good strategy, good tactics being sometimes distinct from sound strategy.

If planning systems had some method of using strategies in a way similar to that of humans, the following seems to be two immediate benefits: (1) many problems with subgoal interactions could be avoided, if the strategies gave either explicit orderings of the subgoals or criteria for ordering them; (2) the "combinatorial quagmire" could be navigated more efficiently, since a predetermined strategy can often automatically exclude many possible operator applications. It remains an open research issue, however, exactly how this to be done.

4.2.3 Fuzzy Goals

It is obvious that in KRS many concepts and goals are not easily expressed in terms of sequences of operations and literals. We will refer to such goals and concepts as being "fuzzy" with respect to the domain. (The term "fuzzy" has no direct connection to the notion of fuzzy sets or fuzzy logic, although it seems likely that important relations could be discovered among these areas.) These fuzzy concepts are useful in human planning and need to be incorporated into planning systems that deal with realistic domains.

Different types of fuzzy concepts may exist, but one characteristic shared by many examples is the notion of an activity or a process occurring over an extended period of time. These concepts cut across state descriptions and do not lend themselves to expression in terms of the truth of predicates in a single state description.

There is another sort of fuzziness as well: the fact that fuzzy concepts often do not seem to be definable by any single set of characteristics or criteria. This generality, vagueness, or "fuzziness" of fuzzy concepts, although a liability as far as incorporation into SL frameworks is concerned, makes them useful in planning in realistic environments.

4.2.4 Fuzzy Algorithms

Any mechanism for using strategy (or doing global planning) must be able to cope with epistemological nondeterminism. It must deal with the uncertainties and incomplete knowledge of real-world environments. A rigid, prefabricated sequence of primitive actions executed subsequent to the creation of the

sequence of actions will not do; the creation of the plan must interact with the environment as the plan is created (i.e., planning and execution must be interleaved [Sacerdoti 79; McDermott 78; Goldkind 81]). The notion of a "fuzzy algorithm" is proposed as a device for incorporating both strategy and flexibility into a planning system.

One should distinguish between fuzzy algorithms and fuzzy statements of algorithms, such as high level descriptions (natural language), flow charts, pseudocode, etc. These descriptions do not specify the algorithm completely with respect to every level of detail needed for implementation. In the case of an ordinary (not fuzzy) algorithm, it must always be possible to completely specify the algorithm in terms of well-defined primitives (for some real or virtual machine). Any process instantiating the algorithm must produce the outputs specified by the algorithm for given inputs [Knuth 68]. Moreover, the algorithm can be filled out to this level of detail before being implemented and before any particular inputs are "given." *A fuzzy algorithm differs in that it is impossible to specify every detail of the algorithm in advance of execution; also, there is no guarantee that the implementation and execution of the fuzzy algorithm will produce the desired results.*

4.2.5 Plan Schemas

A plan schema (if we adhere to the by now almost "traditional" view of a plan as a sequence of actions) determines not just one particular sequence of actions, but rather, a set of sequences of actions each of which is a possible instantiation of the plan schema. This is consonant with the idea of having FA incorporate strategy into the planning process. Strategy need not determine an exact sequence of primitive actions used to carry out the plan (although in simple cases, it might). It serves instead to constrain the primitive actions in various ways, the exact particular actions being a matter of tactics, not strategy. As a simple example of a plan schema, consider:

(GOAL Insure-safety-of-mission
(SUBGOALS (1 Detect-threat)
(2 Respond-to-threat)))

Notice that if the plan schema were to specify subgoals that could, in turn, be specified completely, we would have been returned rather quickly to a standard sort of problem reduction. Some or all of the subgoals referred to by a plan schema must be fuzzy concepts. Several advantages of FPS are immediately apparent. First, we are not limited to combinations of primitive operators at our higher levels of abstraction. In addition, we are not limited to decomposition by logical connectives as the only form of problem decomposition. Having a predetermined strategy can often reduce the need to search through the space of primitive operators.

So far, this is fairly straightforward and could be accommodated by minor alterations to existing theories. A SCRIPT, for example, could be construed as a plan schema (see also the notions of plans and goals presented in [Shank and Abelson 77]).

However, given that a planning system is to employ FPS, and that FPS contain fuzzy concepts (for flexibility and generality) that indicate the subgoals associated with a particular goal, we come to a seemingly difficult problem: if the fuzzy concepts are used as subgoals, and fuzzy goals, by hypothesis, cannot be easily defined or specified, what good does it do the system to have a FPS? The goal of insuring the safety of a mission can be broken down into the subgoals of detecting and then responding to a threat, but how does this get the system to a point at which it can take some kind of action to detect the threat?

Part of the answer is that the system somewhere has information about each of the subgoals in the FPS. Each subgoal is a fuzzy goal that cannot be completely specified yet needs to be somehow specified in order to be used by the system. We choose to use another FPS to specify it, for example:

```

(GOAL Detect-threat
  (SUBGOALS (1 Find-enemy-weapons)
            (2 Identify-nature-of-threat))

```

So that each of these further subgoals can be a fuzzy concept, each is also specified by a FPS. Because the reduction into subgoals must terminate at some point, we must have ordinary plan schemas (and perhaps even individual actions) or "degenerate" FPS somewhere at the bottom of this hierarchy of FPS.

This presents us with another problem. If a partial ordering can be traced through this hierarchy of plan schemas, which terminate at leaf-nodes that are not fuzzy (i.e., determinate or completely specifiable), then the various subgoals throughout the hierarchy are not "fuzzy goals" (since they can, by a process of reduction, be completely specified). On the other hand, we are faced with an infinite proliferation of subgoals if the process does not terminate somewhere with something definite.

The answer lies in the following addition to the basic idea of a plan schema: an FPS consists of a plan schema (whose subgoals are fuzzy goals) plus heuristics for instantiation. For example:

```

(GOAL Insure-safety-of-mission
  (SUBGOALS (1 Detect-threat)
            (2 Respond-to-threat ))
  (HEURISTICS (a (SEQUENCE 1 2 3)) ...)...)

```

By allowing arbitrary subgoal orderings to be specified, we accomplish part of the goal of allowing for strategy with regard to subgoal interactions, the heuristics embodying much of the strategy component. The orderings can, in principle, be much more complicated, for example:

```

(SEQUENCE 1 {7 5 6} {4 3} 2 8 9) or
(SEQUENCE 1 (COND [(ANY {7 5 6})(3 4)]
  [T (2 4 {7 6} 5)]))

```

where set braces, '{' and '}' indicate independent subgoals to be achieved in any order (or concurrently) and 'COND' has its usual meaning. Also, the possibility exists of having the sequence specified less directly in terms of other FPS.

This ability to deal with subgoal orderings is an important capability, but the most important source of fuzziness and flexibility for FPS lies in the ability to specify arbitrary heuristics. Strategy is concerned with relations between (sub)goals, and it makes good sense to allow heuristics to determine these relationships in a flexible way, rather than simply serving to guide the search through a space of primitive operators (as per the traditional role of heuristics).

4.3 METAPLANNING AND STRATEGIES

We have already discussed the importance of metaplaning. Some aspects of meta-planning are already clearly capturable by FPS given the description of the last section; others require further explanation and are marked with an asterisk (*).

[Wilensky 81a] notes several important features of meta-planning:

1. Everyday planning situations often involve various goals that can interact in complicated ways. If both

A and B are subgoals of one particular FPS, P, then it is possible (as mentioned previously) for the heuristics of P to mediate potential interactions or conflicts between A and B. The interesting case, however, is where this is not true, so that A and B are either completely independent except for the overlap which produces the interaction or else the interrelation of A and B is not foreseen prior to actually detecting the interaction during planning. In this case, there will be no one particular FPS with heuristics that already deal specifically with the combination of A and B; a planning system should be flexible enough to deal with cases like this. The solution advocated by Wilensky and others is to allow the system to deal with the interaction as a new planning problem. To deal with the problem, the system can have "knowledge about planning" (in order to decide how to plan). Wilensky refers to this process as meta-planning. If these meta-problems can be formulated as goals (albeit meta-goals), they can be solved using the same general methods that the system normally uses (without, for example, having to consult special purpose critics).

The notion of FA accommodates this idea easily, since all that is required is that there be FPS for dealing with various types of interactions (or any other meta-problems). That is, we should include FPS that deal with planning as subject matter.

2. In most planning systems, high-level goals are simply handed to the planner, often in the form of a problem to be solved or a state to be reached. A semi-autonomous planner (e.g., an exploratory robot out of contact with its home base) must be able to infer its goals based on its overall mission together with its situation. This can be accomplished by FPS as described in the last section. Wilensky's example is of a system for maintaining a nuclear reactor. The robot is in charge of sustaining the generation of power, keeping the floors clean, preventing meltdowns, cleaning up dangerous spills, and maintaining itself. Usually most of these goals have no effect. For example, the robot is not concerned with cleaning up a spill until one occurs.

It seems desirable then, to design a planner capable of recognizing situations in which these tasks should be performed. This could be accomplished with a single FPS having each of these as subgoals and heuristics for determining which subgoal to execute next (in a sense, FPS are already "meta" plans with respect to their subgoals).

3. Meta-planning knowledge can be used for both planning and understanding. Wilensky's version of meta-planning

requires that something be represented declaratively in order to qualify as meta-planning knowledge. This is somewhat narrow if construed as a definition (since there is no reason in principle to exclude the procedural use of meta-knowledge about planning), but his point is well taken. For natural language understanders to cope with many uses of language, they must be able to understand the goals and intentions both of speakers and of agents spoken about. If the knowledge about how to resolve conflicts between goals (or any other meta-planning knowledge) is embedded somewhere inside the control structure of the system, then it is practically impossible for the system to utilize this knowledge in explaining, understanding, or reasoning about the goals and behavior of other systems.

FPS, while not tied to a particular representational scheme (at this point), were conceived as a generalization and elaboration of declarative SCRIPT-like representations (e.g., [London 1978b]). Thus, the FPS can be seen as declaratively representable. It remains a question for further investigation if and how well the heuristics for instantiation can be captured in a declarative representation; this however, is the intention of the present writer. The first stages of investigation should involve a search for such a declarative representation for heuristics which satisfies the other constraints on FPS (that it can serve as a meta-language for itself--see [Perlis 1980; Haas 1982] for some initial progress along these lines).

4. Meta-planning knowledge allows for more flexibility in dealing with cases where no solution can be found. This is similar to (1) if instead of unforeseen interactions, we consider a complete failure of some sort. Wilensky's point is that if meta-knowledge is used in the first place in the formulation of the goals which failed, then it can be used again to reformulate the goals or to try to plan around the problem. This is not possible if the system has no knowledge about planning (since then it can have no knowledge about planning failures). FPS can achieve this same effect in exactly the same way as suggested in (1), viz., have FPS for error recovery. This can be done in an ad hoc way within particular FPS for particular applications and also by having meta-FPS which deal with more general questions. (See McDermott 78; Wilensky 81a] on "themes" and how they give rise to meta-goals.)

Aside from these important effects of meta-planning mentioned by Wilensky and others, there are some additional benefits to be achieved by a particular sort of meta-planning within the framework of FPS:

5. Meta-planning can be useful for knowledge acquisition and learning [Davis 82]. If FPS can deal with other FPS as objects (i.e., from a meta-level), then a system can have various (meta)FPS which tell it how to acquire, construct, delete, alter, and manipulate FPS. Some work has been done along these lines by [Davis 82], and this is an important area of current research; one of the bottlenecks for expert and knowledge-based systems is the problem of how to get the "knowledge" into the system. Related research is underway in the area of intelligent interfaces to database systems. The idea that ties these areas together is a data base interface which has (meta)knowledge of the data base schemas and can interrogate the user to obtain data and construct instances of the schemas. An FPS is a more complicated sort of schema than most data bases handle, so a planning system using FPS could go beyond the basic idea of acquiring knowledge about objects and fitting the knowledge into existing schema. There is also the more exciting possibility of altering existing schema and acquiring new ones (including meta-FPS). This has been explored to some extent by [Davis 82] but FPS offers the additional possibility of changing the schema for acquiring schema. Every activity (with the exception of some small kernel) could be governed by the use of appropriate FPS, including the activity of acquiring new schema and altering old schema. If arbitrary schemas can be altered, then even the schemas controlling schema change can be altered. Thus, research into the abilities of such a system to learn is possible.

The following initial set of constraints on the formalism will be revised and augmented as an ongoing part of research in this area. The representation must be:

1. Able to accommodate widely diverse domains.
2. Interpretable both procedurally and declaratively.
3. Able to express subprocess orderings, possible concurrency, and interprocess communication.
4. Amenable to interactive knowledge acquisition.
5. Capable of acting as meta-language for itself; e.g., express meta-FPS that indicate how to alter existing FPS, acquire new FPS, etc.

4.4 KRS IMPLEMENTATION

In KRS, MITRE implemented a version of FA known as "strategies." The above approach was deemed appropriate for dealing with the problems of planning and replanning in the multi-package tactical air domain for two reasons: (1) combinatorial complexity; and (2) the problems faced in replanning are precisely those which meta-planning is designed to deal with. With the move out of the KNOBS single mission domain to a scenario where multiple packages interacted and the focus was on replanning, it became quite clear that the system needed to do more than search through a problem space defined by alternative choices for slot fillers.

The following paragraphs explain the relations between planning, meta-planning, replanning, and strategies, and then describe the implementation.

4.4.1 Replanning, Meta-Planning, and Strategy Selection

Definition of Meta-Planning: planning whose primary concern is the production of plans for manipulating other planning processes or the plan-structures they produce.

Definition of Replanning: planning whose primary concern is the alteration, with minimum perturbations to existing plans, of previously constructed plan structures in response to difficulties or changes in expectations.

Thus, replanning is a type of meta-planning.

The implementation of strategies in KRS uses the ZETALISP flavor mechanism; however, this is largely a matter of convenience. The strategy language was under development and subject to constant revision; hence, the more flexible and easily alterable flavor definitions were used in preference to the more rigid ZETALISP structure definitions. While more efficient, ZETALISP definitions require, for example, extensive recompilation if the order of occurrence of two instance variables is changed. Strategies do not make use of the most important aspect of Zetalisp flavors; viz. the ability to mix various flavors. Only one flavor is used in the strategy implementation (the "Strategy" flavor). Individual strategies differ from one another only in the values of their instance variables (note that we use the term "strategy" both to refer to the "real" strategy described by the flavor instance and also to refer to the description (the flavor instance) itself, allowing context to establish the referent).

4.4.2 KRS Strategy

The description of the KRS strategy implementation is divided into two parts: the declarative content and the procedural interpretation explaining the PROCEDURAL INTERPRETATION.

4.4.2.1 The Declarative Content

The declarative content of the strategies is stored as the values of the instance variables of instances of the STRATEGY flavor. For discussion, we group the instance variables under the following headings:

1. Goals
2. Subgoals
3. Subgoal Relations
4. History
5. Objects
6. Failures

Generally, the information under "Goals" includes the current task or problem along with an English description of what the strategy is designed to accomplish; under "Subgoals" are given subtasks which may help achieve the goal. "Subgoal Relations" contain sequencing information for the related subtasks; under "History" information is kept about the course of the execution of the strategies such as which subgoals succeeded, which failed and were subsequently taken care of, which failed and could not be taken care of and if true information about raised failures when those occur, the instance variables under the heading "Objects" contain information about which objects the strategy will be concerned with and strategies are linked to their parent. "Failures" we have the instance variables which point to the failure handling strategies used when failures or difficulties are reported. We now move to a more detailed discussion of each of these groups.

1. Goals

AKO
STRATEGY-NAME
TYPE
PURPOSE

Explanation:

AKO stands for "a kind of" this instance variable, which is currently not used, was included in anticipation of use by a natural language component, or to tie into any conceptual hierarchy.

STRATEGY-NAME gives the name of the particular strategy described by the current instance of the STRATEGY flavor.

TYPE indicates whether the current strategy has subordinates (in which case it is "not primitive") or whether it contains directly executable LISP code (in which case it is "primitive").

PURPOSE gives a short English description of what the strategy is designed to do.

2. Subgoals

STRATEGIES

STRATEGIES contains a list of all the (sub)strategies that may be used by the current strategy. These are the related strategies which the current strategy may invoke during the course of its activity. They correspond roughly to the notion of "subgoals" in SL planning systems. The syntax looks like

```
strategies ((1 <first-strategy>)(2 <second-strategy>)
            ... (n <nth strategy>))
```

The numbers here have nothing to do with the order in which the strategies are invoked. They serve merely as handy local names for the strategy in question.

3. Subgoal Relations

SEQUENCE

SEQUENCE is where the information about the order of invocation of any substrategies is kept. At the top-level, a sequence is always interpreted as a list of subsequences executed as an implicit AND. That is, a sequence has the form:

```
sequence (<sub-sequence1> <sub-sequence2>
          ... <sub-sequenceN>)
```

and execution of the subsequences is from left to right until one of the subsequences fails. Different types of orderings (those other than the implicit AND) are expressed through the medium of subsequences. Various types of subsequences are allowed: SINGLETON, OR-LIST, CONDITIONAL, LOGICOVER, ALL-LIST, and FORMAT. A short description of each follows:

SINGLETON is the simplest type of subsequence; it consists of a reference to a single strategy to be executed. Its syntax is equally simple; it is a single integer which occurred in the STRATEGIES instance variable above.

OR-LIST corresponds to the usual logical OR. The syntax is

(OR <singleton1> <singleton2> ... <singletonN>)

and execution goes from left to right until one of the singletons succeeds. A decision was made to limit the complexity of the subsequences governed by the OR-LIST to singletons. This was done in order to keep the strategy language readable and easily understood. In fact, it is quite easy to extend the OR-LIST to govern arbitrary subsequences (see www.cba.hawaii.edu/strat/for-list.asp where this is done). This extension opens the door to considerable abuse in the form of arbitrarily deep imbeddings of the OR-LIST.

CONDITIONAL corresponds to the if-then-else construct. It evaluates a condition and then, depending on the result, executes one of two subsequences. The syntax is

(IF <predicate> <subsequence1> <subsequence2>)

with no limit on the complexity of the subsequences other than those imposed by the OR-LIST. The second clause is desired, then, as a way of stating "if my strategy fails, then I will do something to succeed" in the particular place of a subsequence.

LOOPOVER repeats some particular strategy (a singleton subsequence) until it succeeds or until the strategy fails for one of the objects. The syntax is

(LOOPOVER <name of list of objects> <DOING> <singleton>)

ALL-LIST does not correspond to any of the usual logical control strategies. It is used to execute a list of strategies regardless of the success or failure of each of the immediate strategies. The syntax is

(ALL-LIST <singleton1> <singleton2> ... <singletonN>)

Overall success status is determined by applying the user supplied function to the results. For example, if the code THIRD then the success or failure of a strategy is determined by the value of the ALL-LIST.

FORMAT is a degenerate type of subsequence. It is used to print out a list of messages which must be typed precisely at the point between external messages and the next message. The same as the ZETA LISP FORMAT function.

4. History

```
SUCCESS
MOST RECENT SUBORDINATE
MOST RECENT SUBSEQUENCE
UNSUCCESSFUL SUBORDINATES
HANDLED UNSUCCESSFUL SUBORDINATES
SUCCESSFUL SUBORDINATES
SUPERIOR
ADD OBJECTS
REMOVE OBJECTS
ADD OBJECTS
REMOVE OBJECTS
EXTEND REFINEMENT
```

Since SUCCESS is the default function, the user can use the code FIRST to indicate that the first successful strategy is the one to be used.

MOST-RECENT-SUBORDINATE points to the last strategy which was invoked from the current one. There is also a back pointer from that strategy to the current one. (See **SUPERIOR**.)

MOST-RECENT-SUBSEQUENCE refers to the last member of the **SEQUENCE** list to be attempted. (See **SEQUENCE** above.)

UNSUCCESSFUL-SUBORDINATES contains a list of pointers to the subordinates which failed.

HANDLED-UNSUCCESSFUL-SUBORDINATES lists those subordinate strategies which were not "killed by fire" and then were taken care of by a failure handler (i.e., they failed and then succeeded) through that failure handler.

STEEKBACK points to the strategy instance which invoked the current one.

WHEN-CALLED and **WHEN-CALLED** are not currently being used.

WHO-HELPED indicates any failure handler which may have been used in helping to resolve difficulties encountered by the current strategy.

FAILURE-INFO provides a place for subordinates to report back information to their superiors on their failures.

OBJECTS

OBJECTS
BB-LOCATIONS
OBJECT-LOCATIONS
RETURN-VALUE
PARAMETERS
OBJECTS-HANDLED

OBJECTS is a list of local names for objects the strategy manipulates. By convention, all object names begin with the character "o".

BB-LOCATIONS and **OBJECT-LOCATIONS** are two lists which contain information about where to find the object named by the object names in **OBJECTS**. The method for finding objects responds to a "get-object" message. This method returns the object named by the object-name; it checks first locally to see if the object-name names an object which is known by the current strategy. If the object is not known, a "get-object" message is sent to the strategy's superior. If the whole tree of strategies is searched without finding a strategy which recognizes the object-name, then an error condition is signaled, "no object reference to an unknown object". The "get-object" method makes use of information in **BB-LOCATIONS** and **OBJECT-LOCATIONS** which occurs according to the following rules (where capital letters refer to the RELEVANTS).

BB-LOCATIONS is a slot that contains entries of the following form if the object is

known by the strategy:
[object-name] [BB-NAME] [VARIABLE] [variable-name]

known by the strategy:
[object-name] [BB-NAME] [VARIABLE-NAME] [variable-name]

known by the strategy:
[object-name] [SY-SLOT]
[object-name] [frame where object is found]

in a multiple valued slot:

(<object-name> MV-SLOT
<name-of-frame-where-slot-is-found>)

a frame:

(<object-name> FRAME <name-of-frame>)

supplied by a function:

(<object-name> FUNCTION <name-of-function>)

a parameter:

(<object-name> PARAMETER)

locatable by a strategy:

(<object-name> STRATEGY <name-of-strategy>)

OBJECT-LOCATIONS is a similar assoc list such that:

when the object is a:

VARIABLE, no information is needed from the instance variable
OBJECT-LOCATIONS.

VARIABLE-NAME, no information is needed from the instance
variable OBJECT-LOCATIONS.

SLOT value, the name of the slot is given in
object-locations

FRAME, again no information is required from instance
variable OBJECT-LOCATIONS.

FUNCTION, again no information is required from instance
variable OBJECT-LOCATIONS.

STRATEGY, again no information is required from instance
variable OBJECT-LOCATIONS.

PARAMETER, OBJECT-LOCATIONS will have an entry of the form:

(<object-name> <n>)

where n is the number in arg-list of the parameter

RETURN-VALUE is simply a place to store any value the strategy
might want to save as a return value for use by a superior which invoked it,
or by any other strategy

PARAMETERS is a list of any passed values. If no values are explicitly
passed, then the strategy inherits the value of its superior's PARAMETERS.
(All of this takes place in a "before-execute" demon.)

OBJECTS HANDLED points to any important objects which the strategy
operated on.

6. Facades

FAILURE-HANDLER SUBORDINATE-FAILURES

FAILURE-HANDLER contains the name of a strategy for dealing with a failure of the current strategy. One of the main functions of such a failure handler is to post failure information in the FAILURE-INFO instance variable of the current strategy's superior.

SUBORDINATE-FAILURES contains a list which pairs with each subordinate, a failure handler which can be invoked to handle cases where that subordinate does not succeed.

4.4.2.2 Procedural Implementation

How are these various bits of declarative information used; how are they interpreted?

When a strategy instance is sent an execute message, it tries to accomplish its goal by invoking the appropriate subordinate strategies in the appropriate order. Speaking metaphorically: the strategy instance looks at its sequence and follows the instructions there. The strategy instance examines and tries to execute each of the subsequences in its sequence. Once it determines the type of subsequence, it knows in what order to send execute messages to the subordinate strategies in the subsequence, and how to deal with the results.

In detail, the basic cycle initiated by the execute message is as follows:

- 1 Find out what kind of strategy is being dealt with:
PRIMITIVE or NON-PRIMITIVE.
- 2 If PRIMITIVE, then:
 - a Directly execute the strategy (for now this means execute the function given in the STRATEGIES instance variable of the primitive strategy).
 - b If successful (i.e., the function returns not nil), then set SUCCESS to t and return control to the invoker.
 - c If not successful, find an appropriate failure-handler and let it try to handle the failure.
 - d If c fails, just set success to nil.
- 3 If NON-PRIMITIVE, then: loop through each subsequence in the SEQUENCE and for each such sub-sequence:
 - a Find out what kind of subsequence it is (viz., one of SINGLETON, OR-LIST, CONDITIONAL, LOOPOVER, ALL-LIST, FORMAT).
 - b Dispatch to the appropriate handler for the type of subsequence in question. It will send execute messages in the appropriate sequence and take appropriate actions. (Most importantly it will set the success instance variable of the last strategy instance which was sent an execute message to

the appropriate value. This strategy instance is found on the instance variable `most-recent-subordinate`.

- c. Check to see if the last subsequence was successful (by looking at `MOST-RECENT-SUBORDINATE`). If not, then invoke an appropriate failure handler, otherwise, continue processing the subsequences.
- d. If all the subsequences have been executed successfully, then set `SUCCESS` to `t`, otherwise, invoke the appropriate failure handler.

For example: When the mission planner clicks the mouse on `AUTOPLAN` in the top-level window, a search is done on the list `*ALL-STRATEGIES*` to find a strategy definition associated with the name `PLAN-ATO`. This is how the first top-level strategy is selected. (In fact, the instance is stored as the value of another global variable, `TOP-LEVEL`. This is extremely useful for reflecting at any one time available to the system the steps and strategies actually used in accomplishing its tasks. When a new strategy is found, it is used to create an instance of the flavor "strategy" which has its instance variables set according to the definition of `PLAN-ATO`. The instance looks like this:

(Strategy to `PLAN-ATO`), an object of flavor `STRATEGY`, has instance variable values

AKO	STRATEGY
STRATEGY-NAME	PLAN-ATO
TYPE	NON-PRIMITIVE
PURPOSE	"planning the ato"
OBJECTS	(ATO-LIST)
STRATEGIES	(1 PRESCAN) (2 PLAN-A-PACKAGE) (3 POST-ATO-DONE)
SEQUENCE	(1 (LOOPOVER ATO-LIST DOING 2) 3)
SUCCESS	PENDING
MOST-RECENT-SUBORDINATE	unknown
MOST-RECENT-SUBSEQUENCE	unknown
UNSUCCESSFUL-SUBORDINATES	NIL
HANDLED-UNSUCCESSFUL-SUBORDINATES	NIL
SUCCESSFUL-SUBORDINATES	NIL
SUPERIOR	NIL
WHY-CALLED	LEVEL-1
WHEN-CALLED	PLANNING
FAILURE-HANDLER	ATO-FAILURE
SUBORDINATE-FAILURES	PRESCAN-FAILURE
FAILURE-INFO	(PRESCAN-FAILURE)
BELOCATIONS	(ATO-LIST-A-ABOVE-LOC) (ATO-LIST-B-ABOVE-LOC)
OBJECT-LOCATIONS	NIL
RETURN-VALUE	unknown
PARAMETERS	NIL
WHO-HELPED	unknown
OBJECTS-HANDLED	NIL

(Strategy to `PLAN-ATO`)

The first subsequence of this strategy is `ATO`, which corresponds to `PRESCAN` in the list `*ALL-STRATEGIES*` for a strategy definition corresponding to `PRESCAN` which looks like this:

strategy instance is created and MOST-RECENT-SUBORDINATE gets set to point to the PRESCAN strategy instance. This instance is then sent an execute message. PRESCAN, of course, has its own sequence and subordinates, and the basic cycle is repeated many times. Eventually, PRESCAN either succeeds or fails. If it fails, and all failure handlers fail to resolve the problems, then an error condition is signaled. If PRESCAN is successful, then a loopover handler is used to create instances of PLAN-A-PACKAGE (one for each package). After each of the packages has been processed, the execution of the PLAN-ATO strategy continues with POST-ATO-DONE (this simply records the fact that the ATO has now been planned and made available to other parts of the Air Force--so that any planning that now occurs will be considered REplanning). If the entire operation is successful, the results recorded in the top-level PLAN-ATO will look something like this:

<Strategy to PLAN-ATO>, an object of flavor STRATEGY, has instance variable values:

```

NAME              STRATEGY
STRATEGY-NAME     PLAN-ATO
TYPE              NON-PRIMITIVE
PURPOSE           "planning the ato!"
OBJECTS           (ATO-LIST)
STRATEGIES        ((1 PRESCAN) (2 PLAN-A-PACKAGE) (3 POST-ATO-DONE))
SEQUENCE          (1 (LOOPOVER ATO-LIST DOING 2) 3)
SUCCESS           T
MOST-RECENT-SUBORDINATE  <Strategy to POST-ATO-DONE>
MOST-RECENT-SUBSEQUENCE  3
UNSUCCESSFUL-SUBORDINATES  NIL
HANDLED-UNSUCCESSFUL-SUBORDINATES  NIL
SUCCESSFUL-SUBORDINATES  (<Strategy to POST-ATO-DONE>
  <Strategy to PLAN-A-PACKAGE> <Strategy to PLAN-A-PACKAGE>
  <Strategy to PLAN-A-PACKAGE> <Strategy to PLAN-A-PACKAGE>
  <Strategy to PLAN-A-PACKAGE> <Strategy to PRESCAN>)
SUPERIOR          NIL
WHY-CALLED        (LEVEL 1)
WHEN-CALLED       (PLANNING)
FAILURE-HANDLER    ATO-FAILURE
SUBORDINATE-FAILURES  ((1 PRESCAN-FAILURE))
FAILURE-INFO       unbound
RELOCATIONS       (ATO-LIST VARIABLE *ATO-LIST*)
OBJECT-LOCATIONS  NIL
RETURN-VALUE      unbound
PARAMETERS        NIL
WHO-HELPED        unbound
OBJECTS-HANDLED    NIL
Strategy to PLAN-ATO

```

By using the name of the variable TOP-LEVEL, a path can be traced through the execution of the strategy to the very beginning of the solution of the problem. As an illustration, here is an example of a failure handler which a failure has occurred and is handled successfully by a failure handler:

<Strategy to PLAN-A-PACKAGE>, an object of flavor STRATEGY, has instance variable values:

1. **ANALYZE THE PROBLEM** - What PROBLEM are you trying to solve?
2. **CLASSIFY SUBORDINATES** - What are the PROBLEMS you are trying to solve?
3. **ANALYZE THE PROBLEM** - What are the PROBLEMS you are trying to solve?
4. **ANALYZE THE PROBLEM** - What are the PROBLEMS you are trying to solve?
5. **ANALYZE THE PROBLEM** - What are the PROBLEMS you are trying to solve?
6. **ANALYZE THE PROBLEM** - What are the PROBLEMS you are trying to solve?
7. **ANALYZE THE PROBLEM** - What are the PROBLEMS you are trying to solve?
8. **ANALYZE THE PROBLEM** - What are the PROBLEMS you are trying to solve?
9. **ANALYZE THE PROBLEM** - What are the PROBLEMS you are trying to solve?
10. **ANALYZE THE PROBLEM** - What are the PROBLEMS you are trying to solve?

•

WHEN-CALLED:	(PLANNING)
FAILURE-HANDLER:	NIL
SUBORDINATE-FAILURES:	NIL
FAILURE-INFO:	unbound
BB-LOCATIONS:	unbound
OBJECT-LOCATIONS:	unbound
RETURN-VALUE:	unbound
PARAMETERS:	(PKG1005)
WHO-HELPED:	<Strategy to PLAN-ATTACK-FAILED>
OBJECTS-HANDLED:	(OCA1005)
<Strategy to PLAN-ATTACK>	

From this we can tell that OCA1005 was the mission being planned, that the strategy was a primitive strategy, and that the failure was handled by an instance of PLAN-ATTACK-FAILED (this last is indicated by the WHO-HELPED instance variable). The instance of PLAN-ATTACK-FAILED is as follows:

<Strategy to PLAN-ATTACK-FAILED>, an object of flavor STRATEGY, has instance variable values

AKO	STRATEGY
STRATEGY-NAME	PLAN-ATTACK-FAILED
TYPE	NON-PRIMITIVE
PURPOSE	"finding some way to get the mission off the ground after a planning failure"
OBJECTS	(ATO-LIST MISSION)
STRATEGIES	(1 FILL-EXISTING-HOLE) (2 CREATE-HOLE) (3 APPROPRIATE-AIRCRAFT)
SEQUENCE	(OR 1 3)
SUCCESS	T
MOST-RECENT-SUBORDINATE	<Strategy to FILL-EXISTING-HOLE>
MOST-RECENT-SUBSEQUENCE	(OR 1 3)
UNSUCCESSFUL-SUBORDINATES	NIL
HANDLED-UNSUCCESSFUL-SUBORDINATES	NIL
SUCCESSFUL-SUBORDINATES	<Strategy to FILL-EXISTING-HOLE>
SUPERIOR	<Strategy to PLAN-ATTACK>
WHEN-CALLED	LEVEL 1
WHEN-CALLED	PLANNING
FAILURE-HANDLER	NIL
SUBORDINATE-FAILURES	NIL
FAILURE-INFO	unbound
BB-LOCATIONS	ATO-LIST VARIABLE *ATO-LIST* MISSION PARAMETER
OBJECT-LOCATIONS	MISSION
RETURN-VALUE	unbound
PARAMETERS	(OBJECTS)
WHO-HELPED	NIL
OBJECTS-HANDLED	NIL
<Strategy to PLAN-ATTACK-FAILED>	

When the strategy is invoked, it will first try to fill an existing hole, then try to create a hole, and finally try to appropriate aircraft.

CRAFT. This is handled by using an OR-LIST for the sequence. In this case, domain knowledge provides us with enough information to suppose that the order should be as given, viz., from least to most drastic changes to the specifications of existing plans.

4.5 WEAKNESSES OF KRS IMPLEMENTATION OF STRATEGIES

The MITRE implementation of strategies was an attempt to reap the benefits of various prior planning theories, notably Wilensky's theory of metaplaning, without suffering their inadequacies. Unfortunately, strategies themselves proved to have serious limitations.

After two years of experiment, MITRE was forced to conclude that the KNOBS architecture, which also underlies KRS, was not extensible. For the reasons discussed above, metaplaning had to be grafted into the architecture, but it could not be grafted onto the existing program for the following reason. KRS used a generate-and-test model which meant that metaplaning had to be invoked whenever a new date was being proposed, whenever one was rejected, indeed, constantly. However, neither Wilensky's model of metaplaning nor Goldkind's strategies offered a graceful way of "backpedaling" away from a situation in which the planner had made a bad choice. The only way to deal with this was to ensure that the program did not make such a bad choice, and the way to ensure this was to impose constraints with an extraordinary amount of lookahead ability. Writing such constraints took an extraordinary amount of effort and resulted in intricate hand-coded procedural solutions that required considerable development and maintenance time.

Strategies were basically stripped of their flexibility in order to accommodate this. The whole point of FA is that one does not need to specify all the details of the strategy, because one can rely on heuristics to guide the choice of substrategies. However, in KRS, the flexibility of the system was so inflexible that strategies ended up being hand-coded with fixed procedural decisions instead of heuristics. The unsatisfactory result was a system that worked but that lacked theoretical interest and that was very difficult to maintain or to extend.

It is possible that strategies could have been made to work better with another architecture, one that did not rely on constraint lookahead; one cannot tell from the inconclusive nature of the KRS experiments. One thing, however, is clear. KRS took the KNOBS architecture as far as it could go. If the planner to make further progress, one needs to turn to another architecture, one that is not so heavily enmeshed in the system and not imposed upon it from without.

SECTION 5 - CONCLUSION

KRS now takes its place in the evolution of planning research carried out by MITRE and funded by RADC. Its predecessor, KNOBS, represented state of the art planning technology for its time, the late 1970's. KRS represented the final development of the KNOBS approach to planning, and it also served as a bridge to a new approach, the AMPS theory of metaplanning, developed as a successor to KRS. Without KRS there would have been no way of evaluating the constraint based, bottom up, search intensive strategy used in KNOBS. This was true because KRS, for the first time, confronted the kinds of realistic phenomena - multi-mission planning and replanning - that any true planner would have to face. KRS forced its developers to confront the same research issues that other contemporary planning researchers had identified as central: the need for a strategic or metaplanning component, the need for simulation or truth maintenance, the need to prune the search space, the need to produce coherent explanations for program behavior.

KRS showed that the KNOBS architecture could not, in fact, be extended to deal with these issues. KRS was able to plan mission packages, but it was overwhelmingly difficult to modify. The lack of declarativity in the KNOBS design forced KRS developers to expend huge amounts of time extending the system; it also made system generated explanations of behavior impossible. KRS proved that one could successfully use KNOBS as a basis for multi-mission planning and replanning but at the cost of relying on a hand tailored, maintenance intensive, domain specific solution. For beleaguered mission planners, overwhelmed by the demands of realistic offensive counter aircraft missions, KRS may be a solution. It does the job. However, KRS also pointed the way toward a more general solution of a broad class of planning problems, a solution that avoids many of its weaknesses while providing the same strengths.

KRS's greatest value may be that it served as the experimental vehicle that demonstrated that planning research could take a new step forward. By evaluating their experience with KRS, MITRE researchers were able to come up with a new planning theory which led to the AMPS program. This program extends the same ability to plan offensive counter aircraft missions as KRS, but it surpasses KRS in the following ways:

AMPS maintains a declarative knowledge base. This means that it is easy to add information to the planner; the user need not concern himself with the details of AMPS. This also means that AMPS can explain its behavior because each piece of knowledge is annotated with documentation as it is entered. When a user asks AMPS to explain what it has done, AMPS can provide a list of facts along with their documentation.

AMPS provides a partial process for fast replanning.

AMPS handles reasoning about uncertainty. This is an issue that was completely neglected by both KNOBS and KRS, but it is clearly important because in the real world, information is often incomplete or uncertain.

AMPS is easier to use than KRS or KNOBS.

AMPS inherits many of its design elements from KRS. In fact, it is possible to convert a KRS program to an AMPS program with a minimum of effort. This is a very important feature because it allows the knowledge and experience of KRS to be extended to other domains without the need to redevelop KRS.

AMPS developers might not have realized just how important declarativity is. This concept is often ignored in the breach by AI researchers, it was KRS that hammered home its importance for a declarative planning system.

KNOBS, KRS, and AMPS have all developed from the long-term planning research carried out at MITRE and funded by RADC. Each system was at the forefront of planning research for its time. KNOBS produced a system that could use constraints and a bottom up approach to plan a single air mission. KRS used this approach along with a strategic component to plan more realistic air packages. If the current AMPS will be able to make plans for a variety of domains, using the essential mechanisms of KNOBS and KRS in addition to new mechanisms that add flexibility, extensibility, and the power to express complex behavior.

BIBLIOGRAPHY

- Allen, J. F., "Maintaining Knowledge About Temporal Intervals," TR 86, Department of Computer Science, University of Rochester, Rochester, NY, 1981a.
- Allen, J. F., "A General Model of Action and Time," TR97, Department of Computer Science, University of Rochester, Rochester, NY, 1981b.
- Berliner, Hans L., "Some Necessary Conditions For a Master Chess Program," TR CAI-3, August 1973.
- Birnbaum, L. and Selfridge, M., "Conceptual Analysis of Natural Language," In: Computer Understanding, edited by Schank and Riesbeck, Hillsdale, NJ, 1981.
- Brown, R.H., Millen, J.K., and Searl, E.A., "KNOBS The Final Report," 1982 MS6-20, The MITRE Corporation, Bedford, MA, 1986.
- Bruce, Bertram and Newman, Denis, "Interacting Plans," Cognitive Science 2, 1978.
- Carbonell, J. G., "The Counterplanning Process: A Model of Decision Making in Adverse Situations," Rep. Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, 1979.
- Corkill, Daniel D., "Hierarchical Planning in a Distributed Environment," TR 86-1, August 1979, pp. 168-175.
- Cullingford, R. and Pazzani, M., "Word-meaning Selection in Multimodal Language Understanding Programs," IEEE Trans. on PAMI, PAMI-6, No. 4, July 1984.
- Cullingford, R. and Pazzani, M., "Word Meaning Selection in Multimodal Language Processing Systems," TR-82-13, EE&CS Dept., University of Connecticut, 1982.
- Davis, R. and Lenat, D., "Knowledge-Based Systems in Artificial Intelligence," McGraw-Hill, New York, 1982.
- Engelman, C., Berg, C. H., and Bischoff, M., "KNOBS: An Expert Model of a Knowledge-Based Tactical Air Mission Planning System and a Rule-Based Simulation Simulation Facility," Proc. Sixth Inter. Joint Conf. Artificial Intelligence, 1985, pp. 247-249.
- Ersson, E. W., "Translation of Programs from MAC/LSL to INTERLISP," TR 86-1, The MITRE Corporation, Bedford, MA, November 1986.
- Ernst, G. and Newell, A., "GPS: A Case Study in General Problem Solving," in ACM Monograph Series, Academic Press, New York, NY, 1969.

1. Feigenbaum, E., Lecture at Computer Science Department, Washington University, St. Louis, Missouri, 1983.
2. Feigenbaum, E., "Monitored Execution of Robot Plans Produced by STRIPS," *Proc. 10th Conference on Artificial Intelligence*, 71, Ljubljana, Yugoslavia, August 23-28, 1971.
3. Fikes, Richard E., Hart, Peter E., and Nilsson, Nils J., "Learning and Executing Generalized Robot Plans," *Artificial Intelligence* 3, 1972.
4. Gellman, Peter W., *Chess Skill in Man and Machine*, Springer-Verlag, New York, 1977.
5. Gellman, Peter W., "The Abilities of Machines," Doctoral Thesis, Department of Philosophy, University of Rochester, Rochester, NY, 1981.
6. Gellman, Peter W., "Machines and Mistakes," *Ratio*, v. XXIV No. 2, Basil Blackwell, Oxford, England, 1982.
7. Gellman, Peter W., "Early Algorithms, Planning and Problem Solving," Technical Report No. 100, Washington 83-3, Department of Computer Science, Washington University, St. Louis, Missouri, 1984.
8. Gellman, Peter W., "Planning in the Middle Game," in *The Art of the Middle Game*, (eds) S. J. Perlmutter and K. A. Alexander, Penguin Books Inc., Baltimore, Maryland, 1964.
9. Gellman, Peter W., "Planning and Mental Actions," Doctoral Thesis and TR 106, Department of Computer Science, University of Rochester, Rochester, NY 1982.
10. Gellman, Peter W., "The Frame Problem and Related Problems in Artificial Intelligence," in *Artificial Intelligence and Human Thinking*, edited by Elithorn, A., and Jones, D., Jossey-Bass, San Francisco, CA, 1973.
11. Gellman, Peter W., Hayes, Ruth, Frederick, "A Cognitive Model of Planning," *Cognitive Science*, 7, 1983, pp. 275-317.
12. Gellman, Peter W., Fikes, Richard E., Rosenbloom, Stan, and Cammarata, Stephanie, "Modeling Planning as an Incremental, Opportunistic Process," The Rand Corporation, Santa Monica, California, February, 1976.
13. Gellman, Peter W., Fikes, Richard E., "Modeling Planning As an Incremental, Opportunistic Process," *Artificial Intelligence*, 1976.
14. Gellman, Peter W., "Artificial Computer Programming," V. 1, Addison-Wesley, 1988.
15. Gellman, Peter W., "A Formalization of Knowledge and Action for a Multi-Agent Chess System," *Machine Intelligence* 10, 1982.
16. Harkness, John, *Chess for Dummies*, Dover Publications, New York, NY, 1960.

London, Philip E. "A Dependency-Based Modelling Mechanism for Problem Solvers," CS Department, University of Maryland (College Park), TR-589, NSG-7253, November 1977.

London, Philip E. "Approaches to Object Selection for General Problem Solvers," CS Department, University of Maryland (College Park), TR-632, NSG-7253, January 1978a.

London, Philip E., "Dependency Networks as a Representation of Modelling in General Problem Solvers," CS Department, University of Maryland (College Park), Ph.D. Thesis and TR-698, NSG-7253, September 1978b.

Manna, Zohar and Walkinger, Richard, "Studies in Automatic Programming Logic," Elsevier, North Holland, NY, 1974.

McDermott, Drew, "Planning and Acting," Cognitive Science 2, 1978, 71-109.

McDermott, Drew, "Non-Montonic Logic II," Research Report 174, Yale University Department of Computer Science, New Haven, Conn., February 1980.

McDermott, Drew, "A Temporal Logic for Reasoning About Processes and Plans," RR 176, Computer Science Department, Yale University, 1981.

Minsky, M., "A Framework for Representing Knowledge," in Psychology of Computer Vision, ed. Patrick H. Winston, McGraw-Hill Book Co., New York, 1975.

Parment, Dave Richard, "The Way of The Fox, American Strategy in the War for America, 1775-1783," Greenwood Press, Westport, Connecticut, 1975.

Parment, M. and Engelman, C., "Knowledge Based Question Answering," presented at the Conference on Natural Language Processing sponsored by the Association for Computational Linguistics, February 1983.

Reich, Donald, "Truth, Syntax and Reason," unpublished Doctoral Thesis, Department of Computer Science, University of Rochester, Rochester, NY, 1980.

Robinson, G. and Schank, R. C., "Comprehension by Computer: Expectation Based Processing of Sentences in Context," Research Report #78, Department of Computer Science, University, New Haven, Connecticut, 1976.

Robinson, G., "A Logic for Default Reasoning," Technical Report 79-8, University of British Columbia Department of Computer Science, Vancouver, B.C., July 1979.

Robinson, G. and London, Philip, "Subgoal Protection and Unravelling During Plan Execution," Department of Computer Science, University of Maryland, College

- Roberts, R. B., and Goldstein, L. P., "The FRI Manager," MIT AI Lab, Memo 200, September 1977.
- Sacerdoti, Earl D., "Planning in a Hierarchy of Abstraction Spaces," Artificial Intelligence Group, Technical Note 101, SRI Project 3805-2, January 1978.
- Sacerdoti, Earl D., "The Nonlinear Nature of Plans," Artificial Intelligence Group, Technical Note 101, SRI Project 3805-2, January 1978.
- Sacerdoti, Earl D., "A Structure For Plans and Behavior," Elsevier, North-Holland, New York, 1977.
- Sacerdoti, Earl D., "Problem Solving Tactics," IJCAI '79, August 1979, pp. 1-17.
- Schank, R. C. and Abelson, R. P., "Scripts, Plans, Goals and Understanding," Hillsdale, N. J., Lawrence Erlbaum Assoc., 1977.
- Schank, R., and Colby, K., "Computer Models of Thought and Language," W. H. Freeman and Company, San Francisco, CA 1973.
- Schank, R., Goldman, N., Rieger, C., and Riesbeck, C., "MARGIE: Memory, Analysis, Response Generation and Inference on English," IJCAI Proceedings, 1973, pp. 255-261.
- Shortliffe, E. H., "Computer-Based Medical Consultations: MYCIN," American Elsevier, New York, 1976.
- Stefik, Mark, "Planning with Constraints (MOLGEN: Part 1)," Artificial Intelligence 16 (2), 1981, pp. 112-139.
- Stefik, Mark, "Planning and Meta-Planning (MOLGEN: Part 2)," Artificial Intelligence 16 (2), 1981b, pp. 141-170.
- Stefik, Mark, "An Examination of a Frame-Structured System," Proc. Sixth Inter. Joint Conf. on Artificial Intelligence, Tokyo, 1979, pp. 845-852.
- Sussman, G. J., "A Computer Model of Skill Acquisition," Elsevier, North-Holland, New York, 1975.
- Szolovits, P., Hawkinson, L. B., and Martin, W. A., "An Overview of Owl, A Language for Knowledge Representation," MIT/LCS/TM-86, MIT, Cambridge, MA, June 1977.
- TAC Deputy for Information Systems, "CAFMS Functional Description," Directorate of Information Systems Support, August 1985.
- Waldinger, Richard, "Achieving Several Goals Simultaneously," Artificial Intelligence Center, Technical Note 107, SRI Project 2245, July 1975.

Woods, D. A. and Harris, R. H. "Electronic Aids to Pattern Directed Interference Systems." Academic Press, NY, 1978.

Woods, Robert. "Understanding Complex Situations." *ICAI-79*, August 1979, pp. 164-166.

Woods, Robert. "Meta Planning." *First NCAL*, August 1980, pp. 1-6.

Woods, Robert. "Meta Planning: Representing and Using Knowledge About Planning." *Artificial Intelligence and Natural Language Understanding*, Cognitive Science 5, 1981, pp. 19-228.

Woods, Robert. "A Model for Planning in Complex Situations." Electronics Research Laboratory, College of Engineering, UC Berkeley. Memorandum No. UCB/ERI-M81-46, 1981.

Woods, Robert. "Planning and Understanding." Addison-Wesley Inc., Reading, MA, 1985.

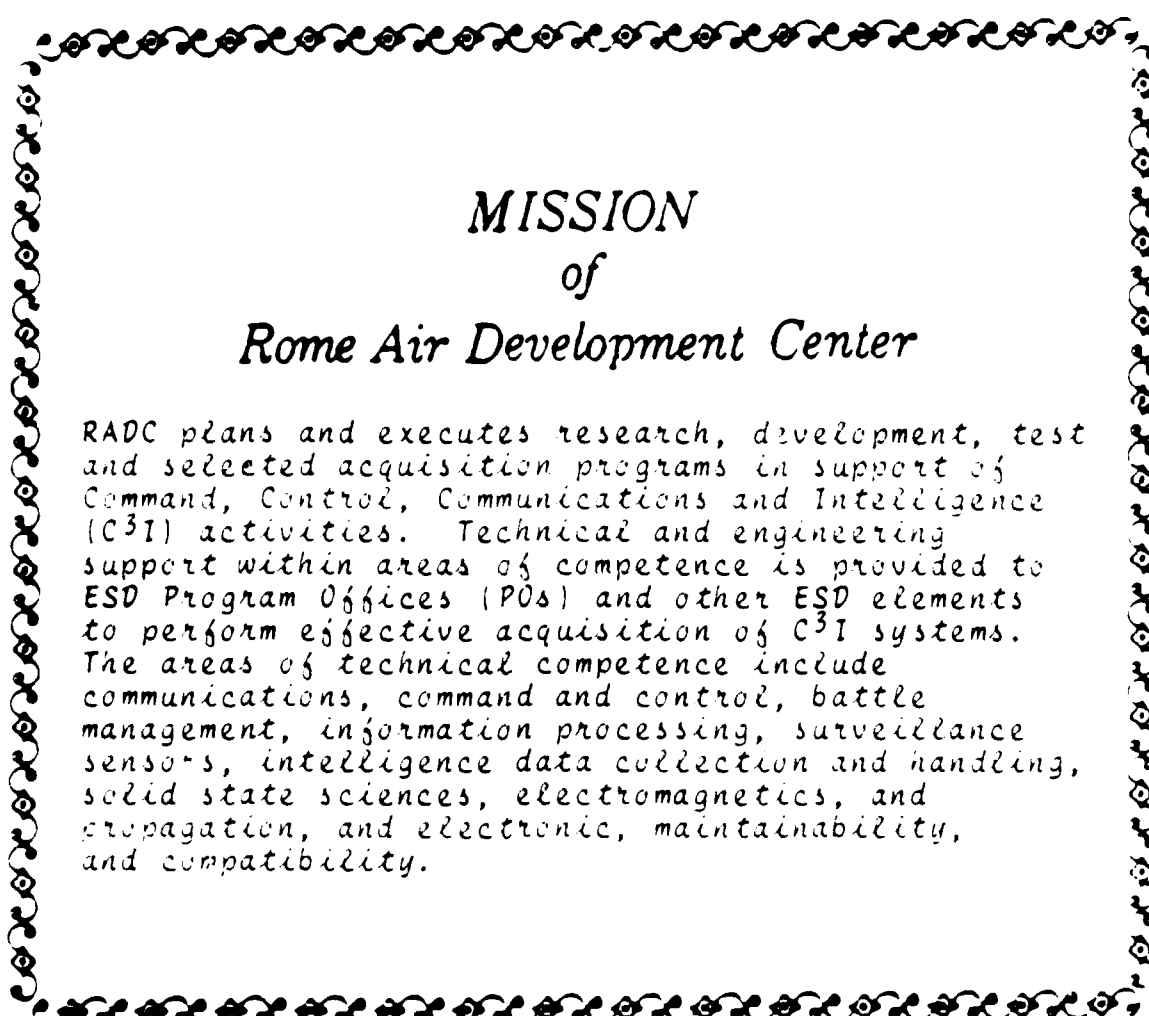
Woods, David. "Using Patterns and Plans in Chess." *Artificial Intelligence* 14 (2), 1980.

Woods, David. "Using Knowledge to Control Tree Searching." *Artificial Intelligence*, 18, 1982.

Woods, A. "Grammar, Meaning & The Machine Analysis of Language." London, 1972.

Woods, W. A. "Transition Network Grammars for Natural Language Analysis." *Comm. ACM*, Vol. 13, No. 10, October, 1970.

Woods, E. C. "Military Strategy: A General Theory of Power Control." Rutgers University Press, New Brunswick, NJ, 1967.



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.

END

8-87

DTIC