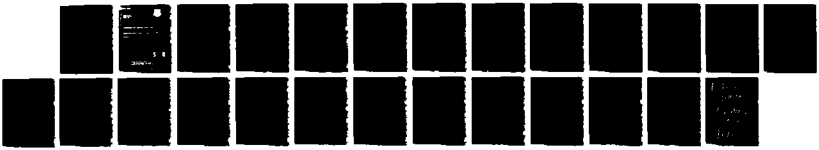
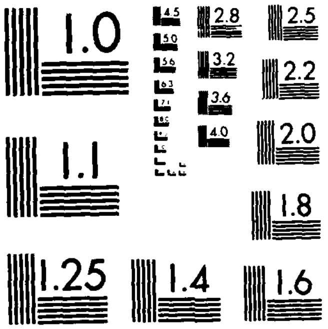
AD-A195 503 COOPERATIVE KNOWLEDGE BASES(U) SOUTHEASTERN CENTER FOR 1/1
ELECTRICAL ENGINEERING EDUCATION INC ST CLOUD FL
W B DAY FEB 88 RADC-TR-88-19 F30602-81-C-0193

UNCLASSIFIED F/G 12/5 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

4

RADC-TR-88-19
Final Technical Report
February 1988

COOPERATIVE KNOWLEDGE BASES

Southeastern Center for Electrical Engineering Education

William B. Day

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

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

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

AD-A195-583

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS N/A
2a. SECURITY CLASSIFICATION AUTHORITY N/A 2b. DECLASSIFICATION / DOWNGRADING SCHEDULE N/A	3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
4. PERFORMING ORGANIZATION REPORT NUMBER(S) N/A	5. MONITORING ORGANIZATION REPORT NUMBER(S) RADC-TR-88-19

6a. NAME OF PERFORMING ORGANIZATION Southeastern Center for Electrical Engineering Education	6b. OFFICE SYMBOL *(If applicable)*	7a. NAME OF MONITORING ORGANIZATION Rome Air Development Center (COES)
6c. ADDRESS *(City, State, and ZIP Code)* Central Florida Facility 1101 Massachusetts Avenue St. Cloud FL 327669		7b. ADDRESS *(City, State, and ZIP Code)* Griffiss AFB NY 13441-5700
8a. NAME OF FUNDING / SPONSORING ORGANIZATION Rome Air Development Center	8b. OFFICE SYMBOL *(If applicable)* COES	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F30602-81-C-0193

8c. ADDRESS *(City, State, and ZIP Code)*
Griffiss AFB NY 13441-5700

10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO	WORK UNIT ACCESSION NO.
62702F	5581	27	P7

11. TITLE *(Include Security Classification)*
COOPERATIVE KNOWLEDGE BASES

12. PERSONAL AUTHOR(S)
William B. Day

13a. TYPE OF REPORT Final	13b. TIME COVERED FROM Jun 87 TO Aug 87	14. DATE OF REPORT *(Year, Month, Day)* February 1988	15. PAGE COUNT 28

16. SUPPLEMENTARY NOTATION
N/A

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP
12	05	

18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
Distributed Knowledge Bases ,
Concurrent Architectures ,
Logic Programming . SDW

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

This report is an examination of the complex issues involved with integrating expert systems from the viewpoint of logic programming. An overview of expert systems and expert system building tools (present and future) is presented, together with three Air Force projects for cooperating expert systems. A summary of a knowledge-based execution system leads to a discussion of how this system can be used to answer many of the integration problems by reduction to the NP-complete problem of allocation. A preliminary static allocation algorithm is proposed.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL John J. Crowter	22b. TELEPHONE *(Include Area Code)* (315) 330-2973 22c. OFFICE SYMBOL RADC (COES)

Previous editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

I. INTRODUCTION

The expert system boom of the past decade has led to a society of autonomous specialized agents. These systems have been applied to problems of interpretation, prediction, diagnosis, design, planning, monitoring, debugging, repair, instruction, and control [Waterman 86]. As this society matures, it becomes necessary to integrate both old and new members into cooperative communities in order to achieve higher levels of abstraction and more sophisticated goals.

This report is an examination of the complex issues involved with integrating expert systems.

In Section II an overview of expert systems and expert system building tools is presented. This includes current work for expanding the inventory of tools to incorporate cooperative systems. This section also describes briefly three Air Force projects which are presently integrating expert systems, viz., the joint Air Force-Navy Project Juniper, the Pilot's Associate, and Expert Systems on Multiprocessor Architectures.

Section III is a summary from [Chung 87c] of a knowledge-based system for parallel processing of logic programs. This system is a prime candidate for a methodology upon which to base distributed, cooperative expert systems. This section discusses the statically allocated architecture, the linguistic enhancements to Prolog which are need for concurrent programming, and the computational model for the system.

Section IV is a discussion of how the system of Section III may be used to confront the problems of integration in a system of experts. These difficulties include acquiring knowledge and heuristics, creating new representations and resolving possible conflicts. The primary emphasis of this section is directed toward the allocation problem of the knowledge base execution system. After the importance of this problem to all phases of the system and its inherent difficulties are posed, a preliminary algorithm is then illustrated with examples.

Section V concludes this report and considers potential future work.

II. EXPERT SYSTEMS: PAST, PRESENT, AND FUTURE

A vast store of expert systems has been created in the last two decades. Although the field of their applications is diverse, an individual expert system is focused on a narrow domain. Construction of new expert systems has been facilitated with the inception of shells and other custom building tools; however, the emphasis continues to be on single, independent systems. Recently, research has begun to address the problems associated with integrating a collection of expert systems. This is the next, logical step in the evolution of a large knowledge base.

In this section we review the fundamentals of the development of an expert system and expert system building tools, including next generation tools. This section ends with summaries of several evolving Air Force projects which are confronting associated issues of integration and distribution.

II.1 Expert Systems Fundamentals.

An expert system is a structured body of knowledge, which is capable of emulating a human expert's power of making decisions. The knowledge can be conceptually divided into a knowledge base, an inference engine, and enhancements for the end-user interface. The knowledge base may consist of the facts and rules (including heuristics) which are specific to the addressed problem domain. The inference engine contains general problem-solving knowledge, typically an interpreter and scheduler. Enchancements include the special system features, which ease interaction with a user. [Waterman 86] defines the construction of an expert system through the sequential tasks of identification, conceptualization, formalization, implementation, and testing with feedback loops from the testing task to the first two tasks through reformalizations, to the third task through redesigns, and to the fourth task through refinements. Since our concern here is with tools for building a system, our emphasis is restricted to the tasks of formalization, implementation, and testing.

The knowledge base is represented either as rules or frames. Basically, the two representations are equivalent and a choice between them is determined by the specific domain. Rule-based representations are constructed from facts and rules, together (optionally) with associated certainty factors. This representation is useful in forward and backward chaining problem-solving. Frame-based representations take the form of either semantic nets or frames. Domains with reliable taxonomies are most appropriate for a hierarchical network of nodes connected by relations, as characterized by semantic nets. The frame data structure works well for domains with stereotypical situations. In a frame each node of a hierarchy has a collection of attributes and associated values, together with attached procedures which monitor value changes. In both frame-based representations, inheritance of attributes minimizes storage and provides a mechanism for inference.

II.2 Expert System Building Tools

Expert system building tools are classified as languages, system-building aids, and support facilities. Different shells combine these components with various options. From another view, expert system building tools contain the three basic units of an expert system, plus they provide an interface to the developer. A complete description and evaluation of current expert building tools in given in [Gevarter 87].

Languages of the tools include general programming languages, such as C, Lisp, or Prolog, as well as knowledge engineering languages. The latter are specialized languages embedded within the system-building aids. The structure of knowledge engineering languages ranges from a skeletal system, which is an existing expert system devoid of its domain knowledge and is applicable only to a restricted class of problems, to a general-purpose system, which can be applied to a variety of different problem areas. Elaborate shells offer the user a choice or combination of languages.

System-building aids are programs which help acquire domain knowledge or guide the design of an expert system. One approach to design has been to construct the system by assembling basic building blocks. Systems like TEIRESIAS provide a method for the system to acquire a human expert's knowledge of domain rules and checks that the rules are complete and consistent.

Support facilities include many enhancements that simplify the system's construction. These include debugging aids, such as tracing the run-time execution or providing break capabilities. Input/output operations can be elaborated via mouse-selected menus or provisions for run-time knowledge acquisition. Typical knowledge base editors support modification, automatically check consistency and syntax, or solicit modifications from the builder through prompts. Finally, the ability of the system to explain its reasoning to a user is critical to a well-developed tool.

Popular building tools are KEE, ART, KnowledgeCraft, and S.1. These sophisticated systems provide numerous options on several different levels to its users. Two new expert building tools that currently have preliminary versions available for test are OPUS from IntelliCorp and ABE from Teknowledge. OPUS is defined by its authors ([Fikes 87])to be a new generation knowledge engineering environment; i.e., it is an evolution of IntelliCorp's KEE. In contrast ABE claims to be a revolutionary tool for intelligent systems, which combines expert/knowledge systems and conventional computer systems. We abstract here the important features of each.

The twin goals of OPUS, as interpreted from the perspective of integrated expert systems, are (1) to provide a common tool base for the developement of components of the system and (2) to integrate these components using new inheritance and default algorithms under the aegis of truth maintenance.

OPUS tools are geared to declarative, object-oriented application models that can perform multiple tasks. This has caused restructuring and clarification of the frame language, including multi-level descriptions of objects, use of relational properties of a slot, and continously active deduction rules. A truth maintenance system is provided to record derivational dependencies and to allow search through a space of alternatives of the domain environment using KEEworlds. The OPUS architecture is designed for compatability with current KEE systems and for support of modular modifications.

ABE's premise ([Erman 86])for improvement in universality; i.e., a multi-level architecture, which offers a choice of different (present and future) modules on different levels is aimed at integrated, large-scale applications featuring re-usable components.

The key design characteristics of ABE are (1) that its various levels are each malleable, (2) that it is easy to learn, (3) that it produces efficent algorithms, which can be iteratively improved, and (4) that it is portable to a variety of machines, including distributed and parallel machines.

The current underlying environment of ABE is the Symbolics LISP machine, Common LISP, and the object-oriented language CORAL. The virtual machine, consisting of MOP (Module Oriented Programming system) and KIOSK (the operating system support of MOP), is created by implementors or systems programmers on a set of communicating modules. It can be mapped onto various hardware/ operating system ervironments. The virtual machine is used by system designers to lay a problem-solving framework (i.e., design choices), such as programming languages, communication protocals and resource allocations. ABE's library now supports blackboards and dataflow frameworks. Above this problem-solving framework is a stratum laid by a tool builder, who supplies new or existing processing modules such as rule interpretation, knowledge maintenance, and explanation. Finally, the knowledge engineer adds structure and control over the modules and their interactions with each other and separate facilities. Currently ABE contains these knowledge engineering tools: MRS, KnowledgeCraft, S.1. Specific domain-dependent information is then filled into the resulting skeleton to create an application system. The first implemented skeletal system was PMR (Plan Monitoring and Replanning) and is independent of a particular application domain. PMR has been used in planning air strike missions and personal travel planning. Two forms of PMR exist, one using the dataflow framework and one using the blackboard framework.

II.3 Air Force Examples of Cooperating Expert Systems

A conceptually simple example of distributed expert systems is the joint Air Force-Navy Project Juniper [Larson 85], which attempts to coordinate an air strike mission using both Air Force and Navy aircraft. The expert systems involved are the Air Force's KRS (Knowledge-based Replanning System) and the Navy's ASPA (Air Strike Planning Advisor).

A number of issues had to be resolved in this integration. First, the two systems were not operating on the same level of expertise. While KRS is a sophisticated, operational planning/replanning system, ASPS had only one of its modules functional, weapons loading. Two other non-technical issues were these: (1) command levels (in the services hierarchies) which were supported by KRS and ASPA were different and (2) because the Navy imposes strict (communication) emissions control aboard carriers when operating in hostile environments, the necessary discourse between the two systems was limited. Other difficulties encountered in devising a realistic scenario are described in [Walter 87].

The two problems were finessed by the decision to develop a Navy version of KRS. In the final version of the scenario, the two versions of KRS interacted directly. ASPA entered the picture only when the Navy KRS had completed negotiations with its Air Force partner; i.e., ASPA received its planning directive and data directly from Navy KRS.

This project was a limited, initial model for communicating expert systems. The final structure was essentially twin expert systems. In transforming KRS to Navy KRS, a bijective mapping of the resources (e.g., carrier instead of airbase) and special values and constraints of the Navy were easily realized. The combined system was then effectively communicating with itself through a dictionary for word replacements.

The Pilot's Associate project, which is one of the primary applications identified by DARPA's Strategic Computing ([Darpa86]), is an ambitious initiative for integrating four expert systems, which collectively assist pilots with managing information, making decisions and performing numerous tasks which will optimize ha pilot's flying and fighting skills. The component systems address system status mission planning, situation assessment, and tactics planning. The intent of the integrated system is to present priority information to the pilot, who acts as the sole arbiter. It is believed that the system's structure promises a high potential for transfer to future aircraft.

The four individual systems have been successfully tested as prototypes, and two competitive contracts have been let for the combined system. At present these duplicate projects are under construction.

Expert systems on muliprocessor architectures is a DARPA-funded project ([Darpa 87]) underway at Stanford. This project addresses many issues which we will examine in Section IV after a description of the knowledge-based execution system in the next section. In particular, the goals of the multiprocessor architecture project include obtaining an increase in speed of two or three orders of magnitude through concurrency, developing a methodology for utilizing concurrency on several levels, deriving a (language) programming setting for concurrency, and defining an architecture which grounds the methodology. We feel that the knowledge-based execution system presents an integrated approach to all these goals, although there are presently no data to justify the increased speed of execution.

In addition to the theorectical and methodological research in their work, the Stanford project is providing numerous experimental results such as contrasting shared and distributed memories, load balancing and evaluating the performance factors of correctness, timeliness, and speedup. Early results support our reasons for choosing a static over a dynamic allocation scheme as discussed in Section III; viz., using a dynamic approach limits scalability and creates a cache consistency problem.

The Stanford project examines two different applications: ELINT, an interpreted system of processed radar signals from aircraft, and AIRTRAC, a system for both spotting smugglers headed for unpoliced airstrips and predicting their flight paths. The hardware system environment is CARE, a simulation of concurrent, communicating processing sites. Either message-passing or shared variables may be used with CARE; see [Brown 86] for further details. The experiments have concentrated on three different frameworks (languages) for implementing ELINT. They are POLIGON, LAMINA, and CAGE.

We elaborate on POLIGON because its overall objectives closely parallel those of our knowledge-based execution system. For example, POLIGON strives for both declarative and procedural semantics, a characteristic of all logic programming. POLIGON's model of parallelism is that an element in the solution space corresponds to a processor. Other requirements of programming languages that POLIGON addresses include these:

1. "the language should provide a tangible method of expressing ideas of the programmer"

2. "the compiler should provide a mapping between the language and the underlying systems, be they hardware or software"
3. "the language should abstract the programmer from its underlying system"
4. "recognize that it is useful to collect one's knowledge of one subject together into one chunk"

Further, language requirements and details of POLIGON are given in [Rice 86].

Other examples of cooperating expert systems are being discussed throughout the AI community, and it's not hard to imagine these applications. For example, the original ASPA goal was to integrate route planning and weapons loading through the subsystem of expert aircraft selection, fuel calculations, and weapons selection. Another idea whose top-level view is planning examines coordinating expert planning on three distinct levels: strategic, reactive, and real-time. One final scenario casts numerous robots, each with its unique capabilities and goals, cooperating for a common goal.

III. A KNOWLEDGE BASE EXECUTION SYSTEM

Very large knowledge-based systems must be able to identify and utilize parallelism, or they will sink under their own weight. The proper viewpoint of identifying parallelism is at the top; i.e., given a multiprocessor, how can we keep as much of the hardware as busy as possible? Since a multiprocessor consists of memory as well as many processing units and since the memory may actually represent over 80% of the actual hardware (at roughly the same cost as the processor [Hillis 85]), it follows that the primary focus of parallelsim should be on using memory in parallel rather than on using processors in parallel.

Given this preamble, we summarize an approach to parallel memory utilization via logic programming; the details are presented in [Chung 87c]. These subsections are included: (1) difficulties of parallelism in logic programs and the AND/OR process model for parallel execution with dynamic allocation, (2) static versus dynamic allocation, (3) parallel language constructs, and (4) a computation model.

III.1 Parallelism and Allocation

Logic programming provides the clause (rule)

```
a(X,Y) :- b(X), c(Y), d(X,Y).
```

with both a declarative interpretation:

a(X,Y) is true if b(X), c(Y), and d(X,Y) are true

and a procedural interpretation:

in order to verify a(X,Y), first verify b(X), c(Y), and d(X,Y).

The order of computation of b, c, and d and the related possibility of AND-parallelism is unspecified, although Prolog choose a left-to-right order. Similarly, when several clauses with identical heads, e.g.,

```
a(X,Y) :- b(X), c(Y), d(X,Y).
a(X,Y) :- e(X), f(Y).
a(X,Y) :- g(X,Y), h(Y).
```

are present, the search rule (and the related possibility of OR-parallelism) is unspecified. Prolog uses top-to-bottom search.

General difficulties in designing a parallel logic language are examined here.

1. Semantic difficulties. There are three interpretations for AND. They are (a) pure AND, (b) AND then, and (c) if then. The OR has two interpretations: (a) pure OR and (b) OR else. The usual solution to these problems is to ignore them as language issues and to consider them solely as procedural problems which can be solved by introducing more control.

2. Binding conflicts. Conflicts which result in binding variables during parallel executions are not always locally detectable or statically analyzable. These conflicts may be several levels deep within a chain. The common solutions are either to execute all clauses and resolve the conflicts or to identify the possible conflicts prior to execution and do them sequentially. Both solutions produce high run-time overhead.

3. Backtracking. Whatever scheme is used for backtracking, it should be made clear what the "most recent" literal is, and do this intelligently and efficiently (the latter two properties are mutually contradictory).

4. Multiple solutions. In OR-exploration, a variable may be bound to many different terms. In such a case it will be necessary to keep a binding environment for each binding or to use only pipeline-parallelism, in which the first solution is used and reported, and the other solutions are stored for possible future use.

5. Problem interaction. The previous four difficulties are not independent. A solution to one problem may solve another or may introduce additional problems.

Previous dynamic allocation schemes have been variations on the AND/OR process model of [Conrey 83]. Here a program is solved by a set of message-communicating processors of two types, AND and OR. They alternate levels as in goal trees with an AND processor at the root and leaves of OR. Messages to an immediate descendant are start, redo, or cancel, and messages to parents are success or fail.

Within an AND-node, one first orders the literals and obtains a dataflow graph. Then for each literal with no inward-pointing arc and which has not yet been assigned, create an OR-node. OR-nodes compute and report one solution at a time, but store alternative solutions found. Backtracking is accomplished using the dataflow graph to a generator (the literal solved first when shared variables are involved). These constant data dependencies are done dynamically and consequently incur high run-time expense.

III.2 Static versus Dynamic Allocation

Measurements of parallelism generally favor a dynamic allocation scheme over a static scheme until one realizes special features of logic programming. Specifically, (1) a dynamic allocation scheme spawns more parallel processes. But this does not matter if the memory is sitting idle. Furthermore, are the processors really busy, making contributions to problem-solving or are they spending most of their time communicating? (2) A dynamic allocation scheme has finer granularity. This is fine if one is working solely on a statement level, but it does not support the modular view. Moreover, finer granularity implies higher system overhead in communications. (3) A dynamic allocation scheme has less of a load-balancing problem. Our view is not concerned with keeping processors occupi'ed but with keeping memory busy. Even if half of our one thousand processors are idle, we may be using five hundred pieces of memory in parallel, each piece executing on its dedicated processor.

Further, a dynamic allocation scheme must limit itself to only one copy of the program; otherwise, the storage costs become excessive. With this limitation, a dynamic allocation scheme must add the initial overhead of dynamically assigning tasks and allocating program segments to the overhead of interprocess communication. A static allocation scheme, on the other hand, has only the overhead of interprocess communication and its one-time initial cost of a static allocation.

Finally, a dynamic allocation scheme must allow for communication between any two processors. In contrast, a static allocation scheme knows the communication requirements prior to execution, and they remain fixed throughout the execution.

From the perspective of execution characteristics of logic programs, a dynamic allocation scheme pales beside its competitor because of these factors:

1. Logic programs have poor locality with the consequences of higher overhead for a dynamic allocation scheme and of a need for partitioning.

2. Administrative information processing forces a dynamic allocation scheme to produce more cumbersome frame manipulations, to remember more details if backtracking is required, and to lose the ideal that any processor can do any task.

3. Binding chains increase communication costs for a dynamic allocation scheme.

4. Interprocess unifications and backtrackings are explosive in number in a dynamic allocation scheme.

From the know]edge base perspective of logic programs, the knowledge base is the number one resource as opposed to conventional programs, whose main resource is the ALU. Knowledge base maintenance puts excessive demands on the host of a dynamic allocation scheme. Only a static allocation scheme can do parallel knowledge base maintenance. The possiblilty of the knowledge base learning under a dynamic allocation scheme looks feeble.

The knowledge base execution system itself consists of a collection of message-communicating nodes, all linked to a special processor, the interface. The interface holds only the allocation map and any special facilities needed to converse with the user. In particular, no part of the program resides within the interface node. Instead the program is allocated initially to the other nodes under a static allocation scheme. The static allocation scheme meets these two requirements:

a. The clauses of a procedural bundle (all clauses with identical heads) are allocated to the same node,
b. For each clause allocated to a node, all procedural bundles which are literals in the clause's body are either allocated to the same node or are allocated to a connected node.

The communication between a required pair of nodes is composed of an input buffer (of sufficient size) and a channel. Messages contain the sender's identification, a message identification, a command (request, redo, cancel, success, fail) and pertinent data.

The operation of the system is based of these three queues, ranked in order: (1) problem-solving, (2) alternative solutions, (3) knowledge base maintenance. At present only the first queue has been implemented.

III.3 Parallel Language Constructs

Three attempts at modifying Prolog to accommodate parallelism have been made. They are known as Concurrent Prolog (see [Shapiro 86]), Parlog (see [Clark 86]), and Guarded Horn Clauses (see [Ueda 86]). We highlight these languages' modifications and introduce our changes to Prolog.

In parallel control the requirement necessary for AND-parallelsim is synchronization; i.e., when two or more literals in the body of a clause share a variable, then *one* must synchronize their execution least a binding conflict result. Concurrent Prolog deals with this issue by annotating (as determined by the programmer) special variables with a "?" . The literals in which these "read-only" variables occur may not be bound in the process of solving the literal, but rather they must be bound prior to the literal's execution or must suspend (or wait) until some other literal, which is in the body and contains the same, unannotated variable, is executed and binds that variable. Thus, the synchronization problem is effected through calls in the body of a clause. Parlog accomplishes its synchronization through the procedure head. Here a mode declaration is added to the Prolog program for each clause in the program. A mode declaration names the predicate and states that each of its arguments is either IN (must be bound) or OUT (must be unbound). The firing of a literal suspends if any one of its IN variables is unbound at the time of execution. Guarded Horn Clauses uses a special suspension rule to solve the synchronization problem.

OR-parallelism involves two requirements: preference and commitment. Preference refers to why one clause is chosen over another in a unification choice. Commitment deletes backtracking. In committed choice nondeterminism, one disallows backtracking once an alternative clause has been selected. This is done in Concurrent Prolog by using the guard, "|", a special symbol added to Prolog, which divides the body of a clause into two pieces. The literals to the left of the guard must be executed and found to be true, before the literals to the right of the guard are allowed to fire. Violations suspend the unification with the clause's head. Because of possible recursion of a unification head and its guard, Concurrent Prolog is restricted to Flat Concurrent Prolog, in which guards are simple tests such as X<0. Parlog also uses guards and limits itself to testing only values of variables that have been declared IN by the mode declaration of the head of the clause. Guarded Horn Clauses uses a second suspension rule (a guard may not bind any variable in the call) to achieve the same effect.

Our knowledge-based execution system deals with AND-parallelsim by extending the mode declarations of Parlog to designate each variable in the head of each clause as IN, OUT, or IN-OUT. The new notation, IN-OUT, allows a variable to be IN if it is bound when the procedure head is invoked or OUT if it is unbound. Mode declarations are mandatory in our new parallel language.

OR-parallelism in our system is supported through pipelining. A node solves a goal, notifies its requestor of the solution, and then begins re-solving the goal. Preference for trying one clause over another with the same head is determined by using a meta-logical notation similar to the mode declaration. Each predicate is declared as either

```
                  in_order(predicate_name, arity)
or
                  no_order(predicate_name, aritv).
```

A no_order choice becomes dependent on the implementation's selection scheme, such as Prolog's top-to-bottom. The ordering allows the node to learn from past experience which alternative leads to a solution and then to adjust its ordering. Although this ordering declaration is optional, it is encouraged since it helps document the programmer's intentions.

Finally, the cut, "!", from Prolog is retained as an extra-logical control mechanism. Since the cut changes the declarative semantics of a clause, its use is discouraged.

III.4 A Computational Model

The initial nodal architecture is that of a conventional processor which can communicate with another node. Each node handles unrelated (from the node's perspective) jobs. If a node does not have the proper clause within its knowledge base (as determined by its piece of the allocation map), it places an ORDER to the correct node. This servant node refers to its requests as JOBs. All messages related to the same job have the same message identification number. A job is composcd of tasks, each started by an incoming message. A task terminates when the node solves the goal, fails to solve the goal, or waits for a servant's response.

Each node is composed of a buffer and a channel for communicating to each linked node. Each node has its individual knowledge base (an unduplicated piece of the program), a proof tree which is composed of frames for the program execution and for the binding histories of each job, and two tables, the job table and the order table, both of which contain information to access the correct frame in a proof tree. Finally, a proving mechanism is provided to each node to manipulate its resources. The forward execution and the backtracking of each node are the fundamentals of the proving mechanism.

After a successful unification in the head of a clause, the literals are active but not yet executed. Instead, they are WAITING, one of five states of literals. A literal is READY to fire when all its arguments satisfy the declaration mode. After a literal is fired, it has status ORDERED, if it sends a request to another node to solve it, or SOLVING, if the original node can solve the literal. When the node or its servant solves the literal, the literal's status become SOLVED.

Each variable in a literal has one of three states: bound, unbound, or held. When a literal is fired, all the unbound variables are held for possible bindings and are therefore committed to this literal. Any other literal in the same clause with the same variable must wait for this variable to be released. A literal is ready to fire when all its IN variables are bound, all its OUT variables are unbound, and no IN-OUT variable is held.

In making a readiness check in the body of a clause from left to right, a decision must be made in the implementation between completing the readiness checks for all literals or firing a literal as soon as it has been determined to be ready. The former choice leads to breadth first execution and has higher potential for parallelism, but it may waste time if siblings to the right of a ready literal all must wait. After a literal is solved the readiness check begins again from the left of the body, but applies only to those literals to the left of the solved literal which are waiting.

Forward execution terminates the readiness checking and literal firings when one of these occurs: (1) the right-most literal is ready, but not locally solvable, (2) the right-most literal is not ready, (3) the current goal unifies with a unit clause (a fact).

Since a literal may be the third one fired but the first one solved, the node must remember the solved order, not the firing order, in order to facilitate backtracking. Solutions to a goal are not only sent back to the requesting node but are also inserted in the knowledge base of the servant node as unit clauses. This is the first level of knowledge base management and also simplifies backtracking.

Shallow backtracking occurs when an attempted unification of variables fails and simply requires trying another clause. Deep backtracking occurs when a node fails to find a clause to solve the current goal and involves a possible combination of three types of deep backtracking.

Intra-clausal deep backtrackig occurs when a failed literal finds a solved sibling. The most recently solved sibling becomes the backtracking point and all other fired literals must be retracted. For ordered literals this consists of canceling orders and resetting the literals to waiting. For solving literals this constitutes undoing bindings and canceling all requests. This form of backtracking is actually naive backtracking on the solved literals.

When a literal fails and no sibling has been solved, inter-clausal deep backtracking occurs. Here one is focusing on the head of the failed clause. All fired siblings must be retracted. One then tries another clause with the same head. This backtracking is always local to the node. Failure in this form requires the servant node also to purge the proof tree and delete this job from the job table.

Finally, inter-process deep backtracking occurs when a literal in the original job fails. This merely requires reporting the failure to the requestor. From this point on, one of the other two forms of deep backtracking will occur.

A more detailed discussion of these mechanisms and an implementation are contained in [Chung 87c].

IV. COOPERATIVE EXPERT SYSTEMS

The management of knowledge bases (in particular, expert systems) is one application that can be handled through our execution system. In this section we examine this aspect by describing how our system can be modified to acquire knowledge and heuristics, create new representations and resolve possible conflicts.

Knowledge can enter our system through three portals. The first opening has already been discussed: each processor, after successfully solving a goal, retains this solution as a fact in its knowledge base.

The second entrance is achieved by augmentation. This includes the addition of a few or many facts and rules by the system user. For example, a user may employ an unused processor to accumulate unrelated clauses. Later, during the next re-allocation of the system (described below), these clauses can be reassigned to processors which are more appropriate. On a grander scale, two or more large knowledge bases can also be combined in this way; i.e., the interface node will hold the entire allocation map and a re-allocation of the entire system will yield a unified, cohesive knowledge base.

The third gateway for knowledge acquisition lies within the processors themselves. Part of each processor is the knowledge base maintenance queue. This queue includes the potential for eliminating redundant information (e.g., from two merged knowledge bases) as well as its own mechanisms for instructing and learning. For example, learning from examples can be built into the knowledge base maintenance queue wherein the test examples are presented to the problem-solving queue of the same processor. This promotes parallel learning for the entire system. Learning by discovery can also be invoked internally; this type of learning at its most fundamental level takes the form of re-organization of the knowledge base.

Learning by discovery also points to the creation of new representations. It is believed that typing and manipulating data structures will ultimately lead to a more human-like learning process. Without some formal form of amalgamating clustered ideas and types, abstraction would be impossible for us humans.

Both acquiring knowledge and creating new respresentations are dependent on the system's ability to re-organize itself. These re-organizations and the initial construction of the network are examples of the allocation problem. We wish to address this fundamental issue more carefully now.

Static allocation of a knowledge base is an NP-complete problem. Although this class of problems cannot be solved (deterministically) in polynomial time, a subclass possesses the property that good approximations can be obtained in polynomial time. The Traveling Salesman paradigm is a familiar example for which one can easily construct an excellent estimate of the optimal path. A mathematical definition and proof of "good approximation" is generally not possible. Rather one must be content with a method that works well for reasonably small test cases which exhibit little pathology.

Consequently, we must limit our search for a static allocation scheme to approximations, and evaluate a particular scheme by comparing it with the random allocation scheme.

For the intelligent system described in Section III, we also envision that over the life-cycle of the system it will be advantageous to re-configure the system many times. A re-configuration will never occur in the middle of solving or re-solving a problem, but only in the knowledge base maintenance cycle. It could be initiated by a subset of the nodes, including the interface. Such re-allocation schemes will be elaborations of the initial allocation, augmented with heuristics and statistics developed since the previous allocation. This re-use increases the importance of the allocation problem.

Another use for an allocation scheme deals with combining expert systems or knowledge bases. The joint system may not necessarily need to create an expert manager. Rather a re-allocation of the combined knowledge bases can potentially be used to eliminate redundancies and restructure the system's network of nodes.

The partitioning of the knowledge base and the corresponding allocation process observe these two principles: (1) clauses of a procedural bundle are allocated to the same processor, and (2) for each clause allocated to a node, all procedural bundles which occur in the clause body are either allocated to the same node or to a connected node. Henceforth, under the first principle, any allocation scheme will be concerned with assigning procedural bundles.

Assume we have N processors available to the system after those needed for infrastructure (such as the interface node) and for special purposes (such as processors dedicated to unification) have been subtracted. Assume there are M procedural bundles in program P. For a large knowledge base N = O(10**4) and M = O(10**5). The total search space for the optimal configuration is then O(N**M). This is an impossible task even if each bundle-to-processor choice is a simple true/false test. Moreover, we have avoided a definition of "optimal". The optimization-definition problem is ill-posed because of lack of universally accepted objective functions and accompanying metrics. Even reducing the question to a minimization of time is not totally accurate since this presupposes that each processor has unlimited storage space. Nevertheless, for a first attempt we make this storage assumption, and define the optimization as an allocation scheme which minimizes programs P's time. It is clear that for whatever initial configuration is chosen, there will generally be some query presented to the system for which the scheme is not optimal. This cannot be avoided with static allocation. Rather we emphasize a scheme which optimizes the "average" query, where we assume that each piece of the code is equally likely to be invoked. If simple counters for the bundles were installed, it would be possible to use this information to weight pieces of code during the next re-allocation. Thus, for the initial allocation we assume that we wish to minimize the time for solving an "average" query.

To summarize, the problem of statically allocating the knowledge base P, which contains M procedural bundles, to N processors is one of finding a "good" approximation, based on heuristic metrics, to the minimization of the program's time for answering an "average" query. Thus, it is definitions for the words "good" and "average" that are the keys to a scheme.

When a literal is fired, it is either in the SOLVING state if the local processor can solve it or it is ORDERED if an external processor can solve it. In determining whether to allocate a procedural bundle to a partially filled processor or to an unused processor, we have to establish a metric for measuring these communication and solution mechanism costs.

For example, if a bundle B is inserted in a partially filled processor Q, there will be no inter-processor communication required of those clauses whoses bodies reference B. However, it will take longer to access B in Q because B and other bundles reside there. Furthermore, Q will halt its readiness checking when it is determined that B is locally solvable. This delays future (to the right in the body of the currently executing clause) ORDERED literals, and thereby inhibits potential parallelism.

On the other hand, placing B in a separate processor Q' introduces the following communication costs: coding the query B in Q, transmitting B from Q to Q', decoding B in Q', unifications leading to a solution for B in Q', coding the solution of B in Q', transmitting the solution of B from Q' to Q, and decoding the solution of B in Q. In addition, there can be delays in each of the two transmissions, as well as a possible severe delay within Q' in solving B. Finally, there is the initial cost of setting up a communication channel between Q and Q', including the creation of the stacks and other data structures needed by the new node Q'.

We consider now a naive scheme for splitting a knowledge base into two knowledge bases, KB1 and KB2. The first step is to weigh all bundles Bi according to these rules:

1. For each unit clause in Bi, add +1 to weight Wi
2. For each non-unit clause in Bi,
   add +1 to Wi for the clause head
   add +1 to Wi for each literal in the body if it is in Bi
   add 0 to Wi for each built-in literal in the body
   add -1 to Wi for each literal in the body if it is not
   in Bi.

For example,

```
    a(x1).    b(y1).    c(z1).    d(x1).
    a(x2).    b(y2).    c(z2).    d(x2).
    a(x3).              c(z3).
    a(x4).
    a(x5).
    a(X) :- b(x), x>10.
    b(X) :- Y is X-1, b(Y).
    c(X) :- a(X).
    d(X) :- b(X).
```

Then

```
        Wa = 5 + (1-1+0) = 5
        Wb = 2 + (1+0+1) = 4
        Wc = 3 + (1-1) = 3
        Wd = 2 +(1-1) = 2.
```

After sorting the bundles by weight, highest first, the bundles are distributed between KB1 and KB2 through scale-balancing. Thus bundle a is assigned to KB1, bundle b is assigned to KB2, and bundle c is also assigned to KB2. The balance now favors KB2 (4+3 > 5) so that bundle d would be assigned to KB1. In general, the heaviest unassigned bundle is added to the lighter knowledge base.

One obvious improvement to this scheme is to re-weigh and re-sort the unassigned bundles after each addition. This affects Wi only where -1 becomes +1 for each literal in the body of the clause if the literal has already been assigned to the lighter knowledge base. In our example, a is again assigned to KB1. Re-weighing and re-sorting bundles b, c, and d produce no change in their order. Note that Wc is still 3 since a is in KB1 and the current lighter knowledge base is KB2. Again bundle b is added to KB2. Now Wc = 3 but Wd = 2 + (1+1) =4 so that bundle d is added to KB2. The balance is tipped, and we insert bundle c into KB1. This revised distribution (bundles a and c in KB1 and bundles b and d in KB2) looks more natural, given the complete set of clauses.

Clearly, the change of -1 to +1 for a literal moving from an external node to the current node is too conservative. The high cost of communication may demand a weight assignment of -10 or even -100 for external literals. A metric must be established for this cost.

This splitting of a knowledge base into two component knowledge bases is the first step in our allocation scheme. The two resulting knowledge bases are then split into four (maybe fewer). This process continues until no resulting knowledge base can be further split or no processor is available.

For a system which re-allocates its knowledge base many times, it will be useful to maintain for each procedural bundle an identification number and a list of pairs. The list for bundle B will consist of all non-B literals in the bodies of the procedures of B. Each element in this list will hold a pair of integers: an identification (of a bundle) number and a frequency count, which denotes the total number of times this non-B literal occurs in bundle B. Since the bundles in this list of B are the only changes that must be made to B's weight, this list provides an efficient method for re-weighing and re-allocating the collection of bundles.

Two distinct approaches to resolving conflicts in a system of cooperative experts are truth maintenance and fuzzy sets. Truth maintenance consists of identifying basic assumptions and annotating each derived goal with a label (a set of these assumptions). Since a goal may be derived in more than one way, it will generally contain a set of labels. While this labeling will eliminate the need for backtracking, the overhead of tagging all goals will become excessive unless some restrictions are made and some backtracking is re-introduced. Typically, the assumptions are split into two disjoint sets. One set holds assumptions that are always true and are always present. These are characteristically fixed or constant assumptions in the domain of discourse; e.g., 2 is even, the sun rises in the east, or aircraft A100 has a maximum range of 2000 miles. The second set is more fluid. An assumption in this set may or may not be believed to be true or false. It is members of this set that dictate a need for truth maintenance.

Several forms of truth maintenance have been implemented in different systems; e.g., KEE allows it when constructing a new expert system. Martin-Marietta has also implemented truth maintenance in a Prolog environment. At some point we would like to incorporate truth maintenance into our knowledge-based execution system, either as part of the problem and re-solving queues or as part of the knowledge base maintenance. Again, our system promotes a parallel implementation of this feature.

Fuzzy set theory has been hailed as a possible break-through in decision-making for expert systems (among other areas); see [Negoita 85]. In negotiating among conflicting views in a language which is semantically imprecise, fuzzy set theory provides some logic for justifying a choice. Although we have not pursued this approach, it remains a potential ally.

V. CONCLUSION

This report addresses the problems of cooperating knowledge bases from the viewpoint of concurrent logic programming. Under our knowledge-based execution system it is possible to inject parallelism into a knowledge base from the top level through minor meta-logical additions to Prolog. Furthermore, independent knowledge bases can be combined into a cohesive unit by using the re-allocation capability of this system. Finally, this system provides a methodology, based on logic, for constructing individual as well as cooperating, intellegent knowledge bases.

The present state of our system for concurrent evaluation of a knowledge base of logic clauses using static allocation consists of an assumed network architecture, linguistic enhancements to Prolog for parallel execution, a computation model, and an ADA simulator of the system. There are three future tasks that must be solved in order to transform this model into a viable system; they are (1) testing the current system, (2) defining an allocation scheme, and (3) data typing.

The present model has been tested for only a few, selected examples. The testing work is intended to extend the system's credibility by recording the response of many, standard logic programs as well as a large knowledge base. The latter example is needed to verify the power of parallel execution.

An allocation scheme must be determined for this system. This scheme and its modifications are required for an initial allocation, for re-allocations throughout the knowledge base's life-cycle, and for merging existing knowledge bases. Some preliminary ideas on this task were presented in Section IV.

Data typing represents an initial investigation into the power of the individual processor's third queue, the knowledge base maintenance queue. Without data typing, it is felt that learning and other forms of maintenance will be defeated by complexity issues. A starting place here is existing work in POPLOG and LOGLISP. It is anticipated that this investigation will lead to efficient ways for the system to learn from examples in parallel.

VI. REFERENCES

[Brown 86]
Brown, H., E. Schoen, and B. Begali, An Experiment in Knowledge-based Signal Understanding Using Parallel Architectures, KSL Report No. 86-69, Computer Science Department, Stanford University, 1986.

[Chung 87a]
Chung, W.-K. and W. Day, The process allocation in parallel interpretation of logic programs, 1987 ACM Computer Science Conference, 422, 1987.

[Chung 87b]
Chung, W.-K. and W. Day, A static allocation approach for parallel execution of logic programs, IEEE Southeastcon '87, 521-524, 1987.

[Chung 87c]
Chung, W.-K. A Knowledge Based System for Parallel Processing of Logic Programs, Ph.D. dissertation, Auburn University, 1987.

[Clark 86]
Clark, K. and S. Gregory, Parlog: parallel programming in logic, ACM Transactions Prog. Sys., Vol. 8, 1-49, 1986.

[Conrey 83]
Conrey, J. The AND/OR Model for Parallel Execution of Logic Programs, Ph.D. dissertation, University of California, Irvine, 1983.

[Darpa 86]
Strategic Computing Second Annual Report, DARPA, Arlington, Virginia, February, 1986.

[Darpa 87]
DARPA sponsored workshop for Knowledge-based Systems, St. Louis, Missouri, April, 1987.

[de Kleer 86]
de Kleer, J., An assumption-based TMS, Artificial Intelligence, Vol. 28, No. 2, 1986.

[Doyle 79]
Doyle, J. A truth maintenance system, Artificial Intelligence, Vol. 12, 231-272, 1979.

[Erman 86]
Erman, L., J. Lark, and F. Hayes-Roth, Engineering Intelligent Systems: Progress Report on ABE, Teknowledge, Inc., Palo Alto, California, 1986.

[Fikes 87]
Fikes, R., OPUS: A New Generation Knowledge Engineering Environment, Phase I Final Report, IntelliCorp, Inc., Mountain View, California, 1987.

[Gevarter 87]
Gevarter, W. The nature and evaluation of commercial expert system building tools, IEEE Computer, May, 1987.

[Hillis 85]
Hillis, W., The Connection Machine, MIT Press, Cambridge, Massachusetts, 1985.

[Negoita 85]
Negoita, C., Expert Systems and Fuzzy Systems, Benjamin-Cummings Publishing Co., 1985.

[Rice 86]
Rice, J., Poligon, a System for Parallel Problem Solving, KSL Report No. 86-19, Computer Science Department, Stanford University, 1986.

[Shapiro 86]
Shapiro, E., Concurrent Prolog: a progress report, IEEE Computer, 44-58, August, 1986.

[Ueda 86]
Ueda, K., Guarded Horn Clauses, in G. Goos and J. Hartmanis (Eds.), Logic Programming '85, Springer-Verlag, 168-179, 1986.

[Walter 87]
Walter, S., K. Benner, and C. Anken, Knowledge-based Replanning Applied to Coordinated-service Mission Planning, RADC Tech. Report, to be published.

[Waterman 86]
Waterman, D., A Guide to Expert Systems, Addison-Wesley, 1986.

☆ U.S. GOVERNMENT PRINTING OFFICE: 1988—511-117/64099

END

DATE

FILMED

9-88

DTIC